Machine Learning[*]

Luke Min

April 1, 2024

Abstract

KNN, perceptrons, gradient descent, linear and logistic regression, linear support vector machines, kernels, kernelization of machine learning models, Gaussian processes, neural networks, decision trees, random forests, bagging and boosting, bias-variance decomposition, PAC learning, k-means clustering, GMM, EM, PCA and dimensionality reduction, diffusion models, transformers and LLMs.

[*]Based on notes from Spring 2024 CIS 5200 at Penn taught by Professors Surbhi Goel and Eric Wong. All errors are mine.

Contents

1 Week 1 **4**
1.1 January 18 - Introduction . . . 4

2 Week 2 **4**
2.1 January 23 - Supervised Learning . . . 4
2.2 January 25 - k-Nearest Neighbors, Curse of Dimensionality . . . 6

3 Week 3 **7**
3.1 January 30 - Linear Models, Perceptron . . . 7
3.2 February 1, 2024 - Gradient Descent . . . 10

4 Week 4 **14**
4.1 February 6, 2024 - Gradient Descent, Empirical Risk Minimization . . . 14
4.2 February 8, 2024 - Linear Regression, Logistic Regression . . . 16

5 Week 5 **18**
5.1 February 13, 2024 - Linear Regression, Logistic Regression . . . 18
5.2 February 15, 2024 - Linear Support Vector Machines . . . 21

6 Week 6 **24**
6.1 February 20, 2024 - Nonlinear Modeling, Kernels . . . 24
6.2 February 22, 2024 - Kernels, Kernel SVMs . . . 27

7 Week 7 **31**
7.1 February 27 - Midterm . . . 31
7.2 February 29 - Gaussian Process . . . 31

8 Week 8 **33**
8.1 March 5 - Spring Break . . . 33
8.2 March 7 - Spring Break . . . 33

9 Week 9 **33**
9.1 March 12 - Neural Networks I . . . 33
9.2 March 14 - Neural Networks II . . . 35

10 Week 10 **37**
10.1 March 19 - Decision Trees . . . 37
10.2 March 21 - Bagging, Random Forests . . . 40

11 Week 11 **42**
11.1 March 26 - Boosting . . . 42
11.2 March 28 - Bias-Variance Decomposition . . . 45

12 Week 12 **48**
12.1 April 2 - PAC Learning . . . 48
12.2 April 4 - Clustering, K-Means . . . 48

13 Week 13 **48**
13.1 April 9 - GMM, EM . . . 48
13.2 April 11 - Dimensionality Reduction, PCA . . . 48

14 Week 14 **48**
14.1 April 16 - TBD . . . 48
14.2 April 18 - TBD . . . 48

15 Week 15 **48**
15.1 April 23 - Diffusion Models 48
15.2 April 25 - Transformers, LLMs 48

16 Week 16 **48**
16.1 April 30 - Introspection: Challenges and Risks of ML 48

1 Week 1

1.1 January 18 - Introduction

Introduction and Overview

- Everything is on Canvas
- CIS 4190/5190: Applied Machine Learning
- CIS 5450 Big Data Analytics - Data Science. Build a machine learning system by yourself, e.g. for a startup
- ESE 5450 Data Mining - Data Science + Math.

You can take CIS 5200 and CIS 5450.
Prerequisites, Homework 0, etc.
Grades

- 6HWs (55%), Midterm (20%), Final (25%)
- Can submit multiple tries for Gradescope. Instant feedback for coding assignments.

What is machine learning?

- Improve **performance**
- At some **task**
- Given **experience**

Types of Machine Learning

- Supervised learning
- Unsupervised learning
- Semi-supervised
- Active / Online / Reinforcement Learning, etc....

2 Week 2

2.1 January 23 - Supervised Learning

Typical ML Pipeline: Data -> Learning Algorithm -> Knowledge

- Data: Experience, Training Data
- Learning: Optimization
- Knowledge: Performance on some task, a model

Choices to make

1. Which Data?
2. Which Algorithm?
3. Pick Hyperparameters, estimate
4. Evaluate or test the model.

Learning Paradigms

1. **Supervised Learning**:

   (a) Training Data $= (x_i, y_i) = (\text{Data}, \text{Labels})$
   (b) Goal is to train a model that gets labels

2. **Unsupervised Learning**:
   (a) No labels. No strict performance metric.

3. **Semi-Supervised Learning**
   (a) Mix of labelled and unlabelled data.

4. **Online Learning**:
   (a) Learning by sequentially taking one data point at a time.

5. **Active Learning**:
   (a) You get to choose the next data point you observe.

6. **Reinforcement Learning**:
   (a) Collect their own data, make their own decisions

Supervised Learning. Our notation for the dataset will be $\mathcal{D}$ which we write as

$$\mathcal{D} = \{(x_1, y_1), \cdots, (x_i, y_i), \cdots, (x_n, y_n)\}$$

Typically the $x_i \in \mathcal{X}$ will be **inputs**, $y_i \in \mathcal{Y}$ the **labels**, and n the number of data points. We call $\mathcal{X}$ the **feature space** (typically $\mathbb{R}^d$) and the $\mathcal{Y}$ the **label space**.

We will also frequently work with the **feature matrix** X and the **label vector** Y which we write as

$$X = \begin{bmatrix} -x_1- \\ -x_2- \\ \vdots \\ -x_n- \end{bmatrix} \in \mathbb{R}^{n\times d} \quad \text{and} \quad Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \in \mathcal{Y}^n$$

Classification Example: Cat vs. Dog. In this case, the dataset $\mathcal{D}$ consists of images of cats or dogs (x_i's) and the labels (y_i). Here the x_i = RGB Pixels. If the images are 1000x1000 pixels and RGB has three values, then $d = 1000 \times 1000 \times 3$. The labels could be written as

$$y_i = \begin{cases} +1 & \text{if dog} \\ -1 & \text{if cat} \end{cases}$$

This is called a (binary) classification problem. In a multi-class classification problem we can say

$$y_i = \begin{cases} (1,0,0) & \text{if dog} \\ (0,1,0) & \text{if cat} \\ (0,0,1) & \text{if otter} \end{cases}$$

Regression Example: Housing Prices. If you want to build a model that predicts housing prices, the dataset $\mathcal{D}$ consists of descriptors of sold houses, and the price the house sold at. x_i could be number of beds or baths, age, size, and y_i could be the price. This is called a regression problem.

Measuring Performance with Error. A model $h : \mathcal{X} \to \mathcal{Y}$ is a function from the feature space to the label space, which outputs a predicted value $\hat{y} = h(x)$. Thus, for each data x_i, we want to compare the predicted value $\hat{y}_i = h(x_i)$ to the actual value y_i. We write the metric used as $\hat{R}$ for **risk** which is

$$\hat{R}_{0/1}(h) = \frac{1}{n}\sum_{i=1}^{n} l_{0/1}(h(x_i), y_i)$$

where the 0/1 loss function

$$l_{0/1}(\hat{y}_i, y_i) = \begin{cases} 0 & \text{if } \hat{y} = y \\ 1 & \text{otherwise} \end{cases}$$

This is usually referred to as **training error of h.** In the case of regression, we can replace the zero one loss with squared loss,

$$l_{sq}(\hat{y}_i, y_i) = (\hat{y}_i - y_i)^2$$

All of machine learning is basically trying to estimate $h(\cdot)$ by minimizing risk.

Generalization Gap. Once you estimate $h(\cdot)$ on training data, we use a separate set of test data to test the the model $h(\cdot)$. The issue is that our model may not generalize to unseen test data. We try to quantify this as the generalization gap which we write as $|\text{Training Error} - \text{Test Error}|$.

2.2 January 25 - k-Nearest Neighbors, Curse of Dimensionality

How do we choose h? If we were to minimize training risk, we could always memorize the entire training data set by

$$h(x) = \begin{cases} y_i & \text{if } x = x_i \\ 1 & \text{otherwise} \end{cases}$$

This extreme example leads to the issue of *generalization.*

Generalization. Suppose we have $(x_i, y_i) \sim \mathcal{P}$ the training data. If in addition we have $(x, y) \sim \mathcal{P}$ then we can define the **true risk** as

$$R(h) = \mathbb{E}_{(x,y)\sim\mathcal{P}}[l(h(x), y)]$$

We want to minimize true risk, not training risk. Now we can write

$$R(h) = \left[\underbrace{R(h) - \hat{R}(h)}_{\text{gap}}\right] + \hat{R}(h)$$

Because we want to minimize true risk $R(h)$, but only observe training risk $\hat{R}(h)$, one thing we could do is set things up so that generalization bound holds, i.e. $R(h) - \hat{R}(h) \le f(n)$ where $f(n) \to 0$ as $n \to \infty$. With infinite data, we can basically close the gap. But we return to this later.

Theoretically, the best we can do is minimize the true risk:

$$\arg\min_h R(h) = h^*$$

We call this h the **Bayes Optimal Classifier**.

Example: Binary Classification. Let's go through a simple example of binary classification. If we know

$$\eta(x) = P(Y = 1|\, x)$$

and use the zero one loss $l_{0/1}$ then we get

$$\begin{aligned} R(h) &= \mathbb{E}_{(x,y)\sim\mathcal{P}}\left[l_{0/1}(h(x), y)\right] \\ &= \mathbb{E}_x\left[\mathbb{E}_{y|x}\left[l_{0/1}(h(x), y)\right]\right] \\ &= \mathbb{E}_x\left[P(Y = 1|\, x)\,\mathbb{I}(h(x) = -1) + P(Y = -1|\, x)\,\mathbb{I}(h(x) = 1)\right] \\ &= \mathbb{E}_x\left[\eta(x)\,\mathbb{I}(h(x) = -1) + (1 - \eta(x))\,\mathbb{I}(h(x) = 1)\right] \end{aligned}$$

In the third line, the 0/1 loss becomes 1 only when h gets the choice wrong.

The choice of $h = h^*$ that minimizes true risk is

$$h^*(x) = \begin{cases} -1 & \text{if } \eta(x) < 0.5 \\ +1 & \text{otherwise} \end{cases}$$

Hence the true risk in this case becomes

$$R(h^*) = \mathbb{E}_x\left[\min(\eta(x), 1 - \eta(x))\right]$$

where we assumed the particular distribution $\eta(x)$ and the loss function $l_{0/1}$.

We call the true risk under Bayes Optimal Classifier the **Bayes Error.**

k-Nearest Neighbors. Suppose we have X's and O's distributed over some space. k-nearest neighbors is an estimator we can use when we *assume that data points close to each other of the same class.*

We take some subset of the training data $N_x \subseteq \mathcal{D}$, which we call the neighborhood of x. Formally we assume that $\forall z \in \mathcal{D}\backslash N_x$, $z' \in N_x$ we have

$$d(x, z) \geq d(x, z')$$

In other words, given some neighborhood of x that we call N_x, all the points in that neighborhood is close to x than any point outside of it.

The **k-Nearest Classifier** is

$$h(x) = \text{mode}\{\, y'|\, (x', y') \in N_x\}$$

One key way this might fail is if the differently labelled data points are clustered together.

Another potential issue is that in high dimensional space, the distances are very far apart. Roughly it might seem like all the points are equally apart, and the KNN CLF spits out something that looks like it was randomly chosen.

Another issue is if our distance function does not capture our distance appropriately. For example, if we take $d(x, z) = \|x - z\|_2^2$ or the Euclidean distance, it might not work very well for some form of data.

3 Week 3

3.1 January 30 - Linear Models, Perceptron

Last lecture, we talked about k-nearest neighbors. The algorithm builds neighborhoods N_x that make the same predictions for labels. Issues may arise in: high dimensions (all points seem equidistant) and selecting an appropriate notion of a distance (e.g. what is a good metric for images?).

Today we discuss **perceptrons**, which we build on to get to deep learning.

Building on k-nearest neighbors, we could instead draw a line that separates the space into differently labelled regions. We call this a **linear predictor**. Perceptron is an algorithm that helps us find this linear predictor.

Lines. Suppose we consider a hyperplane in $\mathbb{R}^d$ defined by

$$w^\top x + b = 0$$

where $x, w \in \mathbb{R}^d$. Note $w^\top x = \|w\| \, \|x\| \cos\theta$.

- If $w^\top x > 0$ then x is on the same side as the region that w is pointing to
- If $w^\top x < 0$, then x is on the other side.

Hence w basically defines a boundary for x's.

Linear Model (w, b). The linear model for classification is given by where

$$h(x) = \operatorname{sgn}\left(w^\top x + b\right).$$

The function

$$\operatorname{sgn}(a) = \begin{cases} +1 & \text{if } a \geq 0 \\ -1 & \text{if } a < 0 \end{cases}$$

gives the "sign" of its argument.

The linear model retains some helpful properties:

Scale Invariance. If we replace the linear model (w, b) with $(\alpha w, \alpha b)$ for some $\alpha > 0$, nothing changes, as all we care about is the angle. So we can set $\|w\| = 1$.

Suppose we have a dataset $(x_i, y_i)_{i=1}^m$. Let $\alpha = \max_i \|x_i\|$ and normalize

$$\tilde{x}_i = \frac{x_i}{\alpha}$$

which gives us the condition that $\|\tilde{x}_i\| \leq 1$.

In practice, we start by scaling the dataset first to satisfy $\|\tilde{x}_i\| \leq 1$, and then find w in that space.

Bias b is useless. Suppose we define new variables as

$$\tilde{x} = \begin{bmatrix} x \\ 1 \end{bmatrix} \quad \text{and} \quad \tilde{w} = \begin{bmatrix} w \\ b \end{bmatrix}$$

which gives us $\tilde{w}^\top \tilde{x} = w^\top x + b = 0$. The tilde transformation adds an extra dimension, and slides the whole plane down by b. Now in this new space, the $\tilde{x}$ goes through the origin. This tells us that we can always redefine the problem so that the relevant subspace goes through the origin.

Margin γ. In words, the margin γ is defined as the minimum distance of x_i to the line defined by w:

$$\gamma := \min_i \left|w^\top x_i\right|$$

This means that given a linear model (w, b) if we have margin γ, there are no points within γ distance of the linear model.

The margin shows how much room you have on either side, and tells you how much space for errors you have. If the classifier has a good margin, then its a good classifier. If the margin is very small, then its a challenging problem.

$y_i w^\top x_i > 0$. If the sign of y_i and $w^\top x_i$ are the same, then we have $y_i w^\top x_i > 0$. In other words, we have $y_i w^\top x_i > 0$ when the prediction is correct. Hence we can use the quantity $y_i w^\top x_i$ to proxy for correctness. If the quantity is very large and positive, then we are very far from the margin, and we are quite sure about that point; if it's very negative, then we place a lot of confidence that it's correct, but it is in fact incorrect.

Perceptron. We first state the assumption behind the perceptron algorithm.

The realizability / Linear Separability assumption: $\exists w_*$ such that $\forall x_i,\ y_i w_*^\top x_i > 0$. $\gamma_{w^*} = \gamma$. This assumes that there is some "perfect" classifier.

Here's the algorithm:

1. Start with $w_0 = 0$
2. For $t = 1, 2, \cdots$ find i such that $y_i w_t^\top x_i \leq 0$. Update $w_{t+1} = w_t + y_i x_i$.
3. Repeat. If None, break.

Why does this algorithm work? Assume $\|x_i\| = 1$. If we get that $y_i w_t^\top x_i \leq 0$, that means we got that point wrong. And as we update,

$$w_{t+1} = w_t + y_i x_i$$

so that

$$\begin{aligned} w_{t+1}^\top x_i &= \left(w_t + y_i x_i\right)^\top x_i \\ &= w_t^\top x_i + y_i \left\|x_i\right\|^2 \\ &= \begin{cases} w_t^\top x_i + \left\|x_i\right\|^2 & \text{if } y_i > 0 \\ w_t^\top x_i - \left\|x_i\right\|^2 & \text{if } y_i < 0 \end{cases} \end{aligned}$$

Closely observe the above under two cases. First, when $y_i = +1$, then, $w_t^\top x_i \leq 0$, and updating the w this way "adds" to w_t so that it is now closer to being positive (which was the correct prediction). Similarly, the opposite holds for $y_i = -1$. Since we got it wrong ("$y_i w_t^\top x_i \leq 0$"), this means $w_t^\top x_i \geq 0$, and we need to bring it down somehow. That's what happens in the updating step.

Adding a wrong point and updating w rotates the whole w to the "correct" place.

Convergence Proof. A measure of closeness to w_* is given by $w_*^\top w_t$. So that can increase if either (1) the angle is getting smaller; or (2) norm of w_t is getting larger. So a better measure is

$$\frac{w_*^\top w_t}{\|w_t\|} = \cos\theta_t$$

Now given an update at t, substitute

$$\begin{aligned} w_*^\top w_t &= w_*^\top \left(w_{t-1} + y_i x_i\right) \\ &= w_*^\top w_{t-1} + y_i \left(w_*^\top x_i\right) \end{aligned}$$

Note $y_i \left(w_*^\top x_i\right) > 0$ since w_* is a perfect classifier. We also know $\left|w_*^\top x_i\right| \geq \gamma$. So we have

$$\begin{aligned} w_*^\top w_t &\geq w_*^\top w_{t-1} + \gamma \\ &\geq w_*^\top w_{t-2} + 2\gamma \\ &\vdots \\ &\geq \underbrace{w_*^\top w_0}_{w_0 := 0} + t\gamma = t\gamma \end{aligned}$$

Thus we conclude

$$w_*^\top w_t \geq t\gamma$$

Now we need to show the denominator is not increasing too fast. The denominator is

$$\begin{aligned} \|w_t\|^2 &= w_t^\top w_t \\ &= \left(w_{t-1} + y_i x_i\right)^\top \left(w_{t-1} + y_i x_i\right) \\ &= \underbrace{w_{t-1}^\top w_{t-1}}_{= \|w_{t-1}\|^2} + 2\underbrace{y_i \left(w_{t-1}^\top x_i\right)}_{<0} + \underbrace{y_i^2 \left(x_i^\top x_i\right)}_{\leq 1} \\ &\leq \|w_{t-1}\|^2 + 1 \end{aligned}$$

Note $y_i \left(w_{t-1}^\top x_i\right) < 0$ because we always pick the "incorrectly predicted points" i to update the w. We have $y_i^2 \left(x_i^\top x_i\right) \leq 1$ because $y_i^2 = 1$ and the norm of x_i is bounded above by 1 by construction.

Continuing we get that

$$\begin{aligned} \|w_t\|^2 &\leq \|w_{t-1}\|^2 + 1 \\ &\leq \|w_{t-2}\|^2 + 2 \\ &\vdots \\ &\leq \|w_0\|^2 + t \\ &= t \end{aligned}$$

Now collecting our two conditions, we have

$$w_*^\top w_t \geq t\gamma \quad \text{and} \quad \|w_t\|^2 \leq t$$

Going back to our angle definition,

$$\begin{aligned} \cos\theta_t &= \frac{w_*^\top w_t}{\|w_t\|} = \frac{\geq \gamma t}{\leq t} \\ &\geq \frac{\gamma t}{\sqrt{t}} = \gamma\sqrt{t} \end{aligned}$$

Now since cosine is bounded, we get that

$$1 \geq \gamma\sqrt{t}$$

so that

$$t \leq \frac{1}{\gamma^2}$$

Caveats. When we define an algorithm, it's good to understand when it fails.

- When we have the realizability assumption fail, then the algorithm fails.
- When we have differently labelled data points in each region ("XOR Problem") it fails the assumption, and the algo doesn't work.

3.2 February 1, 2024 - Gradient Descent

Recap of Perceptron. Recall the perceptron algorithm:

1. Set $w_0 = 0$.
2. For $t = 1, 2, \cdots$
   (a) if $\exists i$ such that $y_i w_t^\top x_i \leq 0$ then $w_{t+1} = w_t + y_i x_i$
   (b) else break

Key takeaways of the perceptron algorithm:

1. We require that $\exists w_*$ such that $\forall i$, $y_i w_*^\top x_i \geq 0$.
2. Define the margin as
$$\gamma = \min_i \left|w_*^\top x_i\right|$$
where we assume that $\|w_*\| = 1$. The guarantee is that if you run the perceptron algorithm, it will terminate after at most $1/\gamma^2$ steps.

Note that it's possible to have $\gamma = 0$, and the guarantee could in theory be a terrible one. This algorithm, just like KNN, relies on the "geometry" of the problem. It's very specific to the structure of the problem, whereas other algorithms are more general purpose, like gradient descent.

Overview. Today we discuss the **gradient descent**, one of the most successful algorithms in machine learning. A lot of modern machine learning applications utilize a version of gradient descent. It works surprisingly well even in contexts where you wouldn't expect it to perform that well. We will see that even the perceptron algorithm that we saw last lecture is the same as gradient descent under appropriately defined objective functions.

What is our goal? Recall that we have the empirical risk

$$\min_h \hat{R}(h) = \frac{1}{n}\sum_{i=1}^{n} l\left(h\left(x_i\right), y_i\right)$$

where h is the model and l is the loss function. We considered two different loss functions. First the zero-one loss for classification:

$$l_{0/1}\left(\hat{y}, y\right) = \begin{cases} 0 & y = \hat{y} \\ 1 & \text{otherwise} \end{cases}$$

And least squares loss for regression

$$l_{sq}\left(\hat{y}, y\right) = \frac{1}{2}\left(\hat{y} - y\right)^2$$

Function Class H What are the function classes we consider? Note for the linear models we had

$$H = \left\{x \mapsto w^\top x + b\right\}$$

so mapping this to the minimization problem we get

$$\min_{w,b} \frac{1}{n}\sum_{i=1}^{n} l\left(w^\top x_i + b, y_i\right)$$

This process of restricting our function class H is called **parametrization**. Now the problem becomes more concrete! For a lot of loss functions, we can optimize this!

General Setup A more general setup would be

$$\begin{aligned} &\min_w F(w) \\ \text{such that} \quad & w \in C \end{aligned}$$

where F is the objective; C is the constraint set; and w is our choice variable. How can we solve this?

Let's pretend for now that F is simple. For instance, for small v we can approximate linearly as in a Taylor approximation:

$$F(w+v) \approx F(w) + \underbrace{\nabla F(w)^\top}_{\text{gradient}} v$$

Here we are assuming that $F(\cdot)$ is *locally linear*. This is the general idea of gradient descent.

Suppose $F(w+v) \leq F(w)$. Then

$$\begin{aligned} F(w+v) &= F(w) + \nabla F(w)^\top v \\ &= F(w) + \nabla F(w)^\top \left(-\eta \nabla F(w)\right) \\ &= F(w) - \eta \left\|\nabla F(w)\right\|^2 \end{aligned}$$

where we used $v = -\eta \nabla F(w)$, which guarantees $-\eta \left\|\nabla F(w)\right\|^2 \leq 0$.

Gradient Descent. Here we present the graident descent algorithm:

1. Set $w_0 \in \mathbb{R}^d$.
2. For $t = 1, 2, \cdots, T$
   (a) $w_{t+1} = w_t - \eta \nabla F(w_t)$
   (b) $F(w_{t+1}) = F(w_t) - \eta \|\nabla F(w_t)\|^2$
3. Stop when $\|\nabla F(w)\| \leq \epsilon$

Note that

$$\begin{aligned} F(w_{t+1}) &= F(w_t - \eta \nabla F(w_t)) \\ &= F(w_t) + \nabla F(w_t)^\top (-\eta \nabla F(w_t)) \\ &= F(w_t) - \eta \|\nabla F(w_t)\|^2 \end{aligned}$$

How should we choose the **learning rate** η? If it is too large, it will "jump over" the minimum point and miss it, and result in **divergence**. If it is too small, the learning will take too long. There are a number of strategies people have taken.

- Take a big first step, and then take many small steps.
- Start with a big η, and then decrease it over time by setting $\eta_{t+1} = 0.99\eta_t$.
- Transformers increase η in the beginning, keep it at that max for a while, and then linearly decrease it.
- etc..

Limitations of Gradient Descent. In what settings will gradient descent struggle to perform?

- Whenever the function $F(\cdot)$ is "flat" for some regions, the algorithm will get struck.
- When we have local minima, it will also get stuck there.

In general, when we stop making "local progress" the algorithm will terminate, i.e. $\nabla F(w_t) \approx 0$. This can happen at (i) local minimum, (ii) local maxima, (iii) saddle point. But what we want is the global minima.

For this class, we will assume there is a global minimum. In case of quadratic functions, this will be true.

In cases of non-differentiable functions, we can use sub-gradients. But for this class, we won't use it. For instance

$$\text{ReLu}(x) = \max\left(w^\top x, 0\right)$$

is used a lot it neural networks.

Perceptron and Gradient Descent Consider the loss function

$$l(\hat{y}, y) = \max(-\hat{y}y, 0)$$

If you run this gradient descent with $\eta = 1$, then you get the perceptron algorithm.

What's happening here? $\hat{y}y > 0$ when we get the correct prediction, in which case the max function is binding and loss is zero. On the other hand, $\hat{y}y < 0$ when we predict wrongly, and the loss is equal to $-\hat{y}y$.

Second Order Approximation. Consider the first and second order approximations:

1. $F(w+v) \approx F(w) + \nabla F(w)^\top v$
2. $F(w+v) \approx F(w) + \nabla F(w)^\top v + \frac{1}{2} v^\top H v$ where $H_{ij} = \frac{\partial^2 F}{\partial w_i \partial w_j}$ is the Hessian

How should we proceed in the second case? Take the derivative with respect to v to obtain

$$\nabla F(w) + Hv = 0$$

or that

$$v = -H^{-1} \nabla F(w)$$

An algorithm that uses this is called the **Newton's Method**. Now we no longer have the learning rate η.

Newton's Method. Here's the actual algorithm.

1. Set $w_1 \in \mathbb{R}^d$
2. For $t = 1, \cdots, T$
    (a) $w_{t+1} = w_t - H^{-1} \nabla F(w_t)$

This works really well if the function is actually close to quadratic. If it's exactly quadratic, then this gets you the solution in a single step.

The cost is that we now have to compute the Hessian, and then invert it. In high dimensions, this becomes quite costly. This is why in many cases we just use the gradient descent in practice.

When to Use Gradient Descent. When is it a good idea to use gradient descent?

- When the function is convex
- When the function is smooth i.e. if v is small then $|f(w+v) - f(w)|$ is small.

Convexity. When we restrict ourselves to a certain class of functions, then we can make some guarantees about the performan ce of gradient descent. **Convex functions** is an important one, and there are a number of definitions. Here's one.

A function F is **convex** if for all $\alpha \in [0,1]$ and for all $w, w' \in \mathbb{R}^d$,

$$F(\alpha w + (1-\alpha) w') \leq \alpha F(w) + (1-\alpha) F(w')$$

If F is differentiable, then F is convex if for all $w, w' \in \mathbb{R}^d$,

$$F(w') \geq F(w) + \nabla F(w)^\top (w' - w)$$

If F is twice differentiable, then we require

$$\nabla^2 F(w) \succeq 0$$

or that the Hessian is the positive semi-definite. The Hessian is the rate of change of the gradient.

4 Week 4

4.1 February 6, 2024 - Gradient Descent, Empirical Risk Minimization

Quick Recap. In gradient descent, the goal is to learn a model $h(\cdot)$ as follows:

1. Pick some parameterized h. For example, take $h(x) = w^\top x + b$ linear.
2. Start with some $w_1 \in \mathbb{R}^d$.
3. For $t = 1, 2, \cdots$, compute $w_{t+1} = w_t - \eta_t \nabla F(w_t)$ for some loss $F(\cdot)$.

This works well under certain assumptions like convexity, which has several definitions:

1. (General) $F(\alpha w' + (1-\alpha) w) \le \alpha F(w') + (1-\alpha) F(w)$ for all $w, w' \in \mathbb{R}^d$ and $\alpha \in [0,1]$.
2. (F is differentiable) $F(w') \ge F(w) + \nabla F(w)^\top (w' - w)$
3. (F is twice differentiable) $\nabla^2 F(w) \succeq 0$ (i.e. PSD), or for all non-zero $v \in \mathbb{R}^d$, $v^\top \nabla^2 F(w) v \ge 0$, or Hessian has non-negative eigenvalues.

A second property to keep in mind is L-smoothness. A function is smooth if its gradient does not change too fast.

1. $\|\nabla F(w) - \nabla F(w')\| \le L \|w - w'\|$
2. $F(w') \le F(w) + \nabla F(w)^\top (w' - w) + \frac{L}{2} \|w' - w\|^2$

The first condition essentially claims that if the inputs w and w' are quite close, then the gradient also has to be sufficiently close. When L is large, then the function is *less smooth.* If $L = 0$ for instance, then the gradient stays the same. We pick the smallest L such that this condition is true for all $w, w' \in \mathbb{R}^d$. If a function is 2-smooth, then it's also 10-smooth.

The second condition basically says the function should lie below a quadratic term. The function is upper bounded by some quadratic term. Constant L modulates how quadratic that second term is.

If we combine convexity and smoothness, it provides both some upper bound and some lower bound, "sandwiching" the function.

Strong convexity is like the opposite of smoothness. It says that the function's gradient should change by at least some rate, and says the function should have some curvature (i.e. it can't just be flat).

Now we present a theorem on the convergence of the gradient descent algorithm:

Theorem 1. *For a L-smooth convex function F, gradient descent with $\eta_t = 1/L$, we have*

$$F(w_{t+1}) - F(w_*) \le \frac{L}{2t} \|w_1 - w_*\|^2$$

where w_* is the optimal value.

This theorem says that our distant to the optimal value at iteration $t+1$, given by $F(w_{t+1}) - F(w_*)$, can be bounded above. Let say we want the bound to be ϵ so that $\frac{L}{2t} \|w_1 - w_*\|^2 \le \epsilon$. Then note that we need

$$t = \frac{L}{2\epsilon} \|w_1 - w_*\|^2$$

The algorithm itself does not depend on w_*, but this theorem gives us some form of theoretical guarantee for convergence.

Proof. First, we begin by showing $F(w_{t+1}) \le F(w_t)$. Use the update condition given by

$$w_{t+1} = w_t - \frac{1}{L}\nabla F(w_t)$$

for $\eta = 1/L$. Smoothness gives us

$$\begin{aligned} F(w_{t+1}) &\le F(w_t) + \nabla F(w_t)^\top (w_{t+1} - w_t) + \frac{L}{2}\|w_{t+1} - w_t\|^2 \\ &= F(w_t) - L(w_{t+1} - w_t)^\top (w_{t+1} - w_t) + \frac{L}{2}\|w_{t+1} - w_t\|^2 \\ &= F(w_t) - \frac{L}{2}\|w_{t+1} - w_t\|^2 \end{aligned}$$

We want to get rid of the gradient term, so we substitute the rearrange update condition $L(w_{t+1} - w_t) = -\nabla F(w_t)$ in the second line. This tells us that we will be decreasing the function at each iteration.

Second, we want to show how close we are getting to the minimizer w_*, i.e. w_t is getting closer to w_*. We use the definition of convexity on w_* and w_t:

$$\begin{aligned} F(w_*) &\ge F(w_t) + \nabla F(w_t)^\top (w_* - w_t) \\ &= F(w_t) - L(w_{t+1} - w_t)^\top (w_* - w_t) \\ &= F(w_t) + \frac{L}{2}\left[\|w_{t+1} - w_*\|^2 - \|w_t - w_{t+1}\|^2 - \|w_t - w_*\|^2\right] \end{aligned}$$

where we complete the square[1] using $2a^\top b = \|a+b\|^2 - \|a\|^2 - \|b\|^2$.

Now we combine the two results:

$$\begin{aligned} F(w_{t+1}) &\le F(w_t) - \frac{L}{2}\|w_{t+1} - w_t\|^2 \\ -F(w_*) &\le -F(w_t) - \frac{L}{2}\left[\|w_{t+1} - w_*\|^2 - \|w_t - w_{t+1}\|^2 - \|w_t - w_*\|^2\right] \end{aligned}$$

where we multiplied the second condition by (-1). We now add the two inequalities to obtain

$$\begin{aligned} F(w_{t+1}) - F(w_*) &\le -\frac{L}{2}\|w_{t+1} - w_t\|^2 - \frac{L}{2}\|w_{t+1} - w_*\|^2 + \frac{L}{2}\|w_t - w_{t+1}\|^2 + \frac{L}{2}\|w_t - w_*\|^2 \\ &= -\frac{L}{2}\|w_{t+1} - w_*\|^2 + \frac{L}{2}\|w_t - w_*\|^2 \\ &= +\frac{L}{2}\left(\|w_t - w_*\|^2 - \|w_{t+1} - w_*\|^2\right) \end{aligned}$$

From here, we proceed by computing telescoping sums. To do that, we sum over $t = 1, \cdots, T$ as follows

$$\begin{aligned} \sum_{t=1}^{T} F(w_{t+1}) - TF(w_*) &\le \frac{L}{2}\left(\|w_1 - w_*\|^2 - \|w_{t+1} - w_*\|^2\right) \\ &\le \frac{L}{2}\|w_1 - w_*\|^2 \end{aligned}$$

since $\|w_{t+1} - w_*\|^2 \ge 0$. Rearranging,

$$\sum_{t=1}^{T} F(w_{t+1}) - TF(w_*) \le \frac{L}{2}\|w_1 - w_*\|^2$$

[1]Let $a = w_{t+1} - w_t$ and $b = -(w_* - w_t)$ so that

$$\begin{aligned} 2a^\top b &= \|a+b\|^2 - \|a\|^2 - \|b\|^2 \\ &= \|(w_{t+1} - w_t) - (w_* - w_t)\|^2 - \|(w_{t+1} - w_t)\|^2 - \|-(w_* - w_t)\|^2 \\ &= \|w_{t+1} - w_*\|^2 - \|w_{t+1} - w_t\|^2 - \|w_t - w_*\|^2 \end{aligned}$$

Recall that $F(w_{t+1}) \le F(w_t)$ for each t, so that as t increases each iteration, F is at least non-increasing. Then, fixing T, we have that for all $t \le T$, we have $F(w_{T+1}) \le F(w_t)$. Hence, we can give an even lower bound

$$TF(w_{T+1}) - TF(w_*) \le \sum_{t=1}^{T} F(w_{t+1}) - TF(w_*) \le \frac{L}{2} \|w_1 - w_*\|^2$$

Thus in the end we get

$$F(w_{T+1}) - F(w_*) \le \frac{L}{2T} \|w_1 - w_*\|^2$$

□

In the case of μ-strong convexity and L-smoothness, we can get even better results (check notes).

Smoothness makes sure you're not overshooting. Convexity ensures that you get to the solution.

4.2 February 8, 2024 - Linear Regression, Logistic Regression

Recall that the gradient descent applies to general algorithms. In particular, we haven't really specified what the objective function is, or what are hypothesis class is. Today, we first link it to the perceptron algorithm, and then generalize to different models.

Quick Recap: Perceptron. Recall our previous setting for the perceptron algorithm: inputs $x \in \mathcal{X} = \mathbb{R}^d$, labels $y \in \mathcal{Y} = \{-1, +1\}$, hypothesis class $x \mapsto \text{sign}\left(w^\top x + b\right)$, loss: $l(\hat{y}, y) = 0$ if $\hat{y} = y$, else 1. ("0/1 loss").

Logistic Regression. Instead of mapping to ± 1, we could map to the continuous interval $[0, 1]$ as a way to estimate probabilities: $x \mapsto \Pr(y = 1 | x) \in [0, 1]$. Previously, we used the $\text{sgn}(\cdot)$ function. Now we can use the **squashing function**

$$\sigma(\cdot) : \mathbb{R} \to [0, 1]$$

resulting in the new hypothesis class

$$x \mapsto \sigma\left(w^\top x + b\right) \in [0, 1]$$

To be more specific, we will be using the **sigmoid function** that is defined as

$$\sigma(v) := \frac{1}{1 + e^{-v}}$$

note that

- $\lim_{v \to -\infty} \sigma(v) = 0$
- $\sigma(0) = 1/2$
- $\lim_{v \to \infty} \sigma(v) = 1$

We could multiply this by a constant to steepen or flatten the function. The steeper it is, harder it is for the gradient descent algorithm to converge.

One reason to do this (rather than the perceptron) would be to make our function continuous, so that we could apply the gradient descent method. Another reason is to handle the noise in the data. Recall that the perceptron algorithm assumes that the data is perfectly separable, which may not hold in many cases. This way, we can interpret the output as a probability, and handle cases where things are not perfectly separable.

Logistic Loss Function. Note that we can no longer use the 0/1 loss, since that will output a loss of 1, unless your predicted value is exactly equal to the label (i.e. $\hat{y} = y$). Our output of the sigmoid function will never be exactly 0 or 1 – this will only happen in the limit. So we need another function.

We will use something called the **logistic loss** as

$$l(\hat{y}, y) = \begin{cases} -\log(\hat{y}) & \text{if } y = 1 \\ -\log(1 - \hat{y}) & \text{if } y = -1 \end{cases}$$

The logistic loss is an upper bound on the 0/1 loss when graphed on the $l(\hat{y}, y)$ and $y \cdot w^\top x$ space.

Let's try to understand what this function does. Our predicted values are restricted to the interval $\hat{y} \in [0, 1]$. First, suppose $y = 1$. In the correct case where $\hat{y}$ is almost 1, then this loss will be zero; if it is instead we are very wrong and $\hat{y}$ is almost 0, then this loss will be tending to $+\infty$. Second, let's say the correct label is $y = 0$. Then if $\hat{y}$ is also close to zero, then $-\log(1 - \hat{y}) \approx 0$ as well. If $\hat{y} \to 1$, then $-\log(1 - \hat{y}) \to +\infty$.

In general, we would like this loss function to (i) be larger when we are wrong; and (ii) be smaller when we are closer to being correct.

An equivalent way to state this would be

$$l(h(x), y) = \log\left(1 + \exp\left(-y \cdot w^\top x\right)\right)$$

The whole goal of this is to frame this so that you are maximizing the probability of your data.

Probabilistic Perspective. In general, we can take two routes in machine learning: (1) Pick a loss function, and just minimize that loss; or (2) Formulate a probability distribution, and then maximize the probability of seeing my data. Sometimes the two are equivalent, which is going to be the case today. But not always.

Checking the Logistic Loss. Let us consider each case, and confirm that the loss function we wrote down is actually equiavalent.

First, let's say $y = 1$. Then,

$$\begin{aligned} -\log(\hat{y}) &= -\log\left(\left(1 + e^{-w^\top x}\right)^{-1}\right) \\ &= \log\left(1 + e^{-w^\top x}\right) \\ &= l(h(x), 1) \end{aligned}$$

Second, if $y = -1$, then

$$\begin{aligned} -\log(1 - \hat{y}) &= -\log\left(1 - \frac{1}{1 + e^{-w^\top x}}\right) \\ &= -\log\left(\frac{e^{-w^\top x}}{1 + e^{-w^\top x}}\right) \\ &= \log\left(\frac{1 + e^{-w^\top x}}{e^{-w^\top x}}\right) \\ &= \log\left(1 + e^{w^\top x}\right) \\ &= l(h(x), -1) \end{aligned}$$

So this confirms our previous claim.

Maximum Likelihood. Let us denote $\mathcal{L}$ the likelihood function, S the sample of the dataset, and θ the parameters. Then,

$$\begin{aligned}
\mathcal{L}(\theta) &= P(S|\theta) \\
&= \prod_{i=1}^{m} P(x_i, y_i|\theta) && \text{iid assumption} \\
&= \prod_{i=1}^{m} P(y_i|x_i, \theta) \cdot P(x_i|\theta)
\end{aligned}$$

Given some sample S, we want to maximize our probability of observing it. Hence the **maximum likelihood estimator** θ is given by the θ that solves the following:

$$\max_{\theta} \mathcal{L}(\theta)$$

The argmax of the above is equivalent to the argmax of the log-likelihood, which is

$$\max_{\theta} \log \mathcal{L}(\theta) = \max_{\theta} \sum_{i=1}^{m} \left[\log P(y_i|x_i, \theta) + P(x_i|\theta)\right]$$

MLE in Logistic Regression. In the logistic regression case we have that $P(y_i|x_i, \theta) = \sigma\left(w^\top x + b\right)$ and $\theta = (w, b)$. Assume that $P(x_i|\theta)$ is constant. Then,

$$\begin{aligned}
\max_{\theta} \log \mathcal{L}(\theta) &= \log \prod_{i=1}^{m} \frac{1}{1 + \exp(-y \cdot w^\top x)} \\
&= -\sum_{i=1}^{m} \log\left(1 + \exp\left(-y \cdot w^\top x\right)\right)
\end{aligned}$$

5 Week 5

5.1 February 13, 2024 - Linear Regression, Logistic Regression

Quick Recap. There were two views we discussed in the context of logistic regression. The first is the so-called ***loss view***. The 3 steps are:

1. Choose a hypothesis class $\mathcal{F}$
2. Choose a loss function l
3. Optimize using $\min_{f \in \mathcal{F}} \frac{1}{n} \sum_{i=1}^{n} l(f(x_i), y_i)$

For the logistic regression, we chose the class $f(x) = \sigma\left(w^\top x + b\right)$ where the model is parameterized by (w, b); and then we picked the loss function

$$l(\hat{y}, y) = \begin{cases} -\log(\hat{y}) & \text{if } y = 1 \\ -\log(1 - \hat{y}) & \text{otherwise} \end{cases}$$

The second was the ***probabilitic view***, where we proceeded as follows:

1. Pick a probability distribution $P(y|x)$
2. Choose a hypothesis class $\mathcal{F} : P(y|x, \theta)$
3. Maximize the likelihood of observing the data by $\max_{f \in \mathcal{F}} \prod_{i=1}^{m} P(y_i|x_i, \theta)$

Again, in the case of the logistic regression, we made the modelling choice of letting $P\left(y \mid x\right)$ be the Bernoulli distribution with parameter p. Then, the hypothesis class becomes $p = \sigma\left(w^\top x + b\right)$. We can think of the *conditional likelihood* as the probability of data conditional on the parameters:

$$\begin{aligned} P\left(\text{data} \mid \text{parameters}\right) &= P\left(y \mid x, \theta\right) \\ &= \prod_{i=1}^{m} P_i\left(y_i \mid x_i, \theta\right) && \text{independence} \\ &= \prod_{i=1}^{m} P\left(y_i \mid x_i, \theta\right) && \text{identical distribution} \end{aligned}$$

There is an equivalence between the two views. Picking a loss function is kind of like picking a probability distribution $P\left(y \mid x\right)$. And in both cases, we had to choose some hypothesis class $\mathcal{F}$ which restricts our class of models.

Linear Regression. For today, we will take the same approach in the case of the linear regression. However, we will only cover the "loss view" and relegate the probabilitic view to the next homework assignment.

Here's the setup. Our hypothesis class is $\mathcal{F} = \left\{f : f\left(x\right) = w^\top x + b\right\}$. By simply adding an extra dimension, we can subsume the bias term b. Hence for simplicity we do not consider it from here on, and write

$$\mathcal{F} = \left\{f : f\left(x\right) = w^\top x\right\}$$

The feature space $\mathcal{X} = \mathbb{R}^d$ and input space $\mathcal{Y} = \mathbb{R}$, with the squared loss

$$l\left(\hat{y}, y\right) = \left(\hat{y} - y\right)^2$$

Minimizing the Empirical Risk. The empirical risk that we will minimize becomes

$$\hat{R}\left(w\right) = \frac{1}{n}\sum_{i=1}^{n} l\left(\hat{y}, y\right) = \frac{1}{n}\sum_{i=1}^{n}\left(w^\top x_i - y_i\right)^2$$

Let's see if we can write this out in matrix notation:

$$X = \begin{bmatrix} - & x_1 & - \\ & \vdots & \\ - & x_n & - \end{bmatrix} \quad \text{and} \quad Y = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}$$

Then

$$Xw - Y = \begin{bmatrix} - & w^\top x_1 & - \\ & \vdots & \\ - & w^\top x_n & - \end{bmatrix} - \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} w^\top x_1 - y_1 \\ \vdots \\ w^\top x_n - y_n \end{bmatrix}$$

Since $a^\top a = \sum_i a_i^2$, we can take the dot product of $Xw - Y$ with itself to get

$$\hat{R}\left(w\right) = \frac{1}{n}\left\|Xw - Y\right\|^2 = \frac{1}{n}\left(Xw - Y\right)^\top\left(Xw - Y\right)$$

This is a convex function because it is a convex function (i.e. a quadratic) of a linear function of w, so we can apply gradient descent to minimize the objective.

To take the gradient, expand out the multiplication

$$\max_{w} \frac{1}{n}\left[w^\top X^\top X w - 2Y^\top X w + Y^\top Y\right]$$

The first order condition becomes $2X^\top Xw - 2X^\top Y = 0$, which can be rearranged to

$$w = \left(X^\top X\right)^{-1} X^\top Y$$

You can also write out the summation and take partial derivatives from there:

$$\hat{R}(w) = \frac{1}{n}\sum_{i=1}^{n}\left(w^\top x_i - y_i\right)\left(w^\top x_i - y_i\right)$$

The gradient can be then obtained via

$$\begin{aligned}\nabla_w \hat{R}(w) &= \frac{2}{n}\sum_{i=1}^{n}\left(w^\top x_i - y_i\right)\cdot x_i \\ &= \frac{2}{n}\sum_{i=1}^{n}\left(w^\top x_i x_i - y_i x_i\right)\end{aligned}$$

Note we could have stopped at the first line, if we're just applying gradient descent. In this case (but not always), there is an analytic expression for the optimal solution, and we solve for $\nabla_w \hat{R}(w) = 0$:

$$\begin{aligned}\sum_{i=1}^{n} y_i x_i &= \sum_{i=1}^{n} x_i x_i^\top w \\ \implies X^\top Y &= X^\top X w \\ \implies \left(X^\top X\right)^{-1} X^\top Y &= w\end{aligned}$$

which then implies that

$$\hat{Y} = Xw$$

Note that we made a pretty big assumption towards the end, which is that $\left(X^\top X\right)$ is invertible. This is partially addressed by regularization – we will get to this eventually. In addition, if $\left(X^\top X\right)$ is not invertible, then this means that there is a whole space of solutions that fit this criteria – and we don't have just a single, unique solution. We could arrive at any one of those solutions via gradient descent. What regularization does is, it "picks" a solution in this space: in particular, it picks the solution with a smaller norm.

There is one instance where the linear regression is "too powerful": when we have $p >> n$, or when we have lots of features, but not enough observations. For example, we could have $p = n$, and X is invertible. In this case, our previous solution reduces to $w = X^{-1}Y$ and $\hat{Y} = Xw = Y$. This is bad because we just completely memorized the data and our model will not generalize. In very high dimensions, we can find a hyperplane of dimension p that perfectly fits all the data points.

Regularization. Our solution to this issue is **regularization**:

$$\min_w \left\{\hat{R}(w) + \lambda t(w)\right\}$$

where λ is the hyperparameter and $t(w)$ is the regularization term.

Thanks to duality, we can rewrite the above as follows. For each λ, there exists a B such that the following optimization problem has the same solution as the above:

$$\begin{gathered}\min_w \hat{R}(w) \\ \text{such that } t(w) \leq B\end{gathered}$$

In other words, restricting our regularization term $t(w)$ to be less than or equal to B is the same as putting a soft penalty $\lambda t(w)$ on the objective function.

Now because those two are equivalent, we will generally go with the first optimization problem, since gradient descent is easier to apply there.

Examples of Regularization. The first example of regularization is **L2-regularization**, in which case we have

$$t(w) = \|w\|_2^2$$

We also have **L1-regularization**, which is given by

$$t(w) = \|w\|_1^2 = \sum_{i=1}^{d} |w_i|$$

The **Elastic Net** combines the two, as in L1+L2.

Consider L2-regularization. If we think of the constraint set given by $t(w) \leq B$, then this means that solutions to L2 must lie inside a circle. For L1, this means the solution must lie within a diamond shaped region on the w-space (e.g. w_2 vs. w_1 space).

5.2 February 15, 2024 - Linear Support Vector Machines

SVMs were fairly popular until deep learning came along. One way to teach this material is (if you're familiar with linear or quadratic programming) to formulate things in terms of a primal and a dual. Today, we instead take a different approach so that we can just apply gradient descent to it.

We are still in the binary classification setting where $y_i \in \{-1, 1\}$, with model $h(x) = \operatorname{sgn}(w^\top x)$. Suppose we have two classifiers w_1 and w_2, each with margins $\gamma_1 < \gamma_2$. Then in this case, w_2 is a "better" classifier than w_1 since the margin is larger.

If we accept that this larger margin is a good quality to have for our classifier, then we might want an algorithm that always picks the linear classifier with the largest margin. In support vector machines, this is precisely our goal: to find a linear classifier w_* with **maximum margin**.

Margins. Suppose we have a hyperplane x_p and some point x. w is the normal vector to x_p. The distance between point x and x_p is given by the norm of some vector $d = x - x_p$

$$\|d\| = \|x - x_p\|$$

Two things we note are:

1. $d = \alpha w$ (d is parallel to w)
2. $w^\top x_p = 0$ (x_p lies on the hyperplane)

Rearranging d, we get $x_p = x - d$. Using the second property,

$$\begin{aligned} 0 &= w^\top (x - d) \\ &= w^\top (x - \alpha w) \end{aligned}$$

which implies that $w^\top x = \alpha \|w\|^2$ or

$$\alpha = \frac{w^\top x}{\|w\|^2}$$

Nowwe can compute d as in

$$d = \alpha w = \left(\frac{w^\top x}{\|w\|^2} \right) w$$

Then the distance that we are looking for is

$$\|d\| = |\alpha| \|w\| = \frac{|w^\top x|}{\|w\|}$$

To recap, what we did is compute the distance between a plane and a point. Now, if we have some point x_i the distance is

$$\gamma(x_i) = \frac{|w^\top x_i|}{\|w\|}$$

We can now formally define what a margin is. Given a normal vector w and data sample S, we have

$$\gamma(w, S) = \min_i \frac{|w^\top x_i|}{\|w\|_2}$$

Refining the Optimization Problem. The way we will proceed is to first naively come up with an optimization problem whose solution may have the desired properties. Then we will progressively refine it until it is what we want.

One way to compute the w_* would be to choose

$$\max_w \gamma(w, S) = \max_w \min_i \frac{|w^\top x_i|}{\|w\|_2}$$

This is bad. Why? This does not care about any of the labels, and always has a trivial solution, which is to set a hyperplane far away from any of the data points out to infinity.

Fix #1: The fix is to make it a constraint to have w_* correctly classify all the points as in

$$\begin{aligned} &\max_w \gamma(w, S) \\ \text{such that} \quad & y_i w^\top x_i \geq 0 \quad \text{for all } i \end{aligned}$$

where y_i is the label and $w^\top x_i$ is the prediction. If they have the same sign, then the prediction is correct.

Fix #2: This optimization problem does exactly what we want to do; however, the issue is that this optimization is hard to solve. The objective is usually required to be well-behaved. Ideally we want to find a way to simplify the expression $\min_i \frac{|w^\top x_i|}{\|w\|_2}$. The second fix involves noticing that we have a scale invariance: if w is a solution, then αw is also a solution. We then just "define" our normalizing constant to be

$$p := \min_i |w^\top x_i|$$

If we rescale $w' = \frac{w}{p}$, then

$$\begin{aligned} \min_i \left| \left(\frac{w}{p}\right)^\top x_i \right| &= \frac{1}{p} \min_i |w^\top x_i| \\ &= \frac{1}{p}(p) \\ &= 1 \end{aligned}$$

One thing to note is that this constant is specific to w, i.e. $p = p_w$.

Then, our newest optimization problem is to

$$\begin{aligned} &\max_w \frac{1}{\|w\|} \\ \text{such that} \quad & y_i w^\top x_i \geq 0 \quad \text{for all } i \\ & \min_i |w^\top x_i| = 1 \end{aligned}$$

Fix #3: Instead of maximizing the denominator, we minimize the quadratic:

$$\begin{aligned} &\min_w \|w\|^2 \\ \text{such that} \quad & y_i w^\top x_i \geq 0 \quad \text{for all } i && (A1) \\ & \min_i |w^\top x_i| = 1 && (A2) \end{aligned}$$

Fix #4: We still have $\min_i \left|w^\top x_i\right| = 1$ which is not the most tractable.

$$\begin{aligned} &\min_w \|w\|^2 \\ \text{such that} \quad & y_i w^\top x_i \geq 1 \quad \text{for all } i \qquad\qquad (B) \end{aligned}$$

which combines the last two constraints into one.

We now write a short proof that the constraints (A1) and (A2) are the same as the constraint (B). First suppose (A) holds. Then, $y_i w^\top x_i \geq 0$ and $\min_i \left|w^\top x_i\right| = 1$, can be combined to show that for all i $\left|w^\top x_i\right| \geq 1$. Since $y_i \in \{-1, +1\}$, this implies $y_i w^\top x_i \geq 1$.

Second, suppose (B) holds, so that the optimal w_* has the property that for all i, $y_i w^\top x_i \geq 1$. The first constraint (A1) immediately follows since $y_i w^\top x_i \geq 1 \geq 0$. The second constraint (A2) can be shown by contradiction. We can quickly establish that $\min_i \left|w_*^\top x_i\right| \geq 1$, and would like to show a stronger result which is $\min_i \left|w_*^\top x_i\right| = 1$. Suppose on the contrary that $\min_i \left|w_*^\top x_i\right| =: p > 1$. Define another $w_{**} = \frac{w_*}{p}$ so that

$$\|w_{**}\| = \frac{\|w_*\|}{p} < \|w_*\|$$

where we initially assumed that w_* was optimal, but found another w_{**} whose norm is smaller than our optimal w_*. This is a contradiction, and our w_* cannot have been optimal in the first place, and hence $\min_i \left|w_*^\top x_i\right| = p = 1$.

Hard Margin SVM. In the end, we arrive at our final optimization problem:

$$\begin{aligned} &\min_w \|w\|^2 \\ \text{such that} \quad & y_i w^\top x_i \geq 1 \quad \text{for all } i \end{aligned}$$

The solution to this is called the **Hard Margin Support Vector Machine**.

This is called "hard-margin" because we require linear separability via $y_i w^\top x_i \geq 1$. One drawback is that any kind of violation of linear separability will break this system. We require the resulting classifier to be perfect.

How can we address this? Consider the constraint of the problem. We could add some slack ξ_i so that for all i,

$$y_i w^\top x_i \geq 1 - \xi_i$$

and the inequality is easier to satisfy. We can further require a penalty on each of the slack terms, so that the objective becomes

$$\min_w \left\{ \|w\|^2 + C \sum_i \xi_i \right\}$$

thereby each slack costs the objective. Because we don't want to "cheat" by adding negative slack to the objective, we further add a constraint that for all i

$$\xi_i \geq 0$$

Intuitively, any point that is already correctly classified has slack $\xi_i = 0$. If a data point is correctly classfied, but within the margin bound, our $\xi_i \in [0, 1]$. If it incorrectly classified, and outside the margin bound, then at least $\xi_i > 1$.

Soft Margin SVM. Using these slack terms, we arrive at

$$\begin{aligned} &\min_{w, \{\xi_i\}_{i=1}^m} \left\{ \|w\|^2 + C \sum_i \xi_i \right\} \\ \text{such that} \quad & \forall i, \xi_i \geq 1 - y_i w^\top x_i \\ & \xi_i \geq 0 \end{aligned}$$

We combine that last two constraints into

$$\xi_i \geq \max\left(0, 1 - y_i w^\top x_i\right)$$

However, note that we can simplify this further. Notice we will never have $\xi_i > \max\left(0, 1 - y_i w^\top x_i\right)$ at the optimum, because if that were the case, we could set a lower ξ_i and cost us less in terms of the penalty in the objective. Hence we will always have

$$\xi_i = \max\left(0, 1 - y_i w^\top x_i\right)$$

Since we have an equality, we can remove the constraint and substitute into the objective

$$\min_{w, \{\xi_i\}_{i=1}^m} \left\{ \|w\|^2 + C \sum_{i=1}^m \max\left(0, 1 - y_i w^\top x_i\right) \right\}$$

This is the final form of the **Soft Margin Support Vector Machine**.

This turns out to be convex, and we can simply run gradient descent on this. This looks familiar, and reminds us of the linear regression with quadratic in the objective with some regularization term. This term is called the **hinge loss**:

$$l\left(\hat{y}, y\right) = \max\left(0, 1 - \hat{y}y\right)$$

A larger and negative $\hat{y}y$ results in a very large loss. Once you are beyond 1, there is no penalty. 0/1 loss is lower than the hinge loss. (Logistic loss is in between the two?).

6 Week 6

6.1 February 20, 2024 - Nonlinear Modeling, Kernels

Recap: Support Vector Machines We started with some intuitive optimization problem, and whittled it down to what we called the **hard-margin SVM** :

$$\begin{aligned} &\min_w w^\top w \\ \text{such that} \quad &\forall i, y_i w^\top x_i \geq 1 \end{aligned}$$

Allowing for slacks, we also derived the **soft-margin SVM** given by

$$\begin{aligned} &\min_{w, \xi_i} \left\{ w^\top w + C \sum_i \xi_i \right\} \\ y_i w^\top x_i &\geq 1 - \xi_i \\ \xi_i &\geq 0 \end{aligned}$$

where the ξ_i's are the slack variables. We could rewrite this problem using the **hinge loss** terms

$$\min_w \left\{ \underbrace{w^\top w}_{l_2\text{-regularization}} + C \sum_i \underbrace{\max\left(0, 1 - y_i w^\top x_i\right)}_{\text{hinge loss}} \right\}$$

When $C = \infty$, then you will always set the hinge loss term to zero, effectively reverting back to the hard-margin SVM. When $C = 0$, then the regularization term wins. It is good to have an intuitive understanding of what the C parameter does.

Hinge loss penalizes you more the further you away from the prediction. Outliers will be penalized strongly. In addition, unlike the 0/1 loss, it does not just set loss to zero as soon as you get a prediction correctly. It gives a lower loss if you are able to predict with some room for error. (Imagine drawing the hinge loss with $y_i w^\top x_i$ on the x-axis and $\max\left(0, 1 - y_i w^\top x_i\right)$ on the y-axis).

Support Vectors. Why is the above called a support vector machine? A support vector is given by the points on the margin:

$$SV = \left\{x_i : y_i w^\top x_i = 1\right\}$$

Why is this important? The classifier is effectively given just by the points on the margin. If you move the points outside the support vectors, it will not affect the classifier. In other words, the SVM is completely defined by the support vector. The support vector is a "sparse" solution in this sense.

In the case of soft margin SVMs, the support vector includes all the points with the non-zero slacks, although the visualization is a lot cleaner in the hard-margin case.

How do you find the support vectors? Cycle through i's and check for the condition $y_i w^\top x_i = 1$.

Drawback. Linear separability is still a strong requirement to impose on the data, which is a problem.

When did we have a non-linear classifier? K-Nearest Neighbors. The decision boundaries were non-linear in that case. However, one problem then was that the algorithm was not great when the dimensionality was high.

One way we could handle non-linearity is by using **Kernels, non-linear maps**. The idea is that if we are not able to linearly classify data points in the current feature space, why don't we create further non-linear features that we can correctly classify with?

Imagine we have data points we can perfectly classify using distance from the origin. For instance, let's say there's some circle with radius $r = 1$ where all the points inside will be classified one way, whereas all the points outside it will be classfied in another way. Hence, we can come up with some map

$$\begin{aligned} x &\to \phi(x) \\ \mathbb{R}^d &\to \mathbb{R}^D \end{aligned}$$

that can help us achieve this. Let's try the following:

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad \text{and} \quad \phi(x) = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_1^2 + x_2^2 \end{bmatrix}$$

where the last $x_1^2 + x_2^2$ term gives us the distance from the origin.

We can then write

$$\begin{aligned} h(x) &= \operatorname{sgn}\left(w^\top \phi(x)\right) \\ &= \operatorname{sgn}\left(w_1 + w_2 x_1 + w_3 x_2 + w_4\left(x_1^2 + x_2^2\right)\right) \end{aligned}$$

If we were to set $w_1 = -1$,$w_2 = w_3 = 0$, $w_4 = 1$, then we get

$$\begin{aligned} h(x) &= \operatorname{sgn}\left(-1 + x_1^2 + x_2^2\right) \\ &= \begin{cases} +1 & \text{if } x_1^2 + x_2^2 > 1 \\ -1 & \text{if } x_1^2 + x_2^2 < 1 \end{cases} \end{aligned}$$

Choice of ϕ. Picking the $\phi(\cdot)$ map will consist of most of the work. There are some well-known kernel maps that work well. You can try a number of candidates to see which one results in a lower error, and pick that.

Note we can make ϕ more complex. One way to do this is by using **Polynomial Feature**

Map in the following way:

$$\phi(x) = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_d \\ x_1 x_2 \\ \vdots \\ x_{d-1} x_d \\ x_1^2 \\ \vdots \\ x_d^2 \end{bmatrix}$$

where the first entry is the bias term; the next d entries are the **original features**; the rest are the degree 2 terms. The total number of features is given by $1 + d + d^2$. The example here is given for degree $p = 2$.

If we generalize to polynomial degree of p, we get a total of $1 + d + d^2 + \cdots + d^p$ features. As p increases, the increase in dimensions rapidly becomes costly. We will see that there is a way to deal with the curse of dimensionality.

Idea. We can still learn a linear function in $\mathbb{R}^D$. Our solution will be two steps:

1. See if we can compute $\phi(x)^\top \phi(z)$ cheaply. Note we don't have to compute ϕ itself.
2. Check if we can write $w = \sum \alpha_i \phi(x_i)$ for some $\alpha_i \in \mathbb{R}$.

Why does this work? To make a prediction, we need to compute

$$\begin{aligned} w^\top \phi(x) &= \left(\sum_{i=1}^m \alpha_i \phi(x_i) \right)^\top \phi(x) \\ &= \sum_{i=1}^m \alpha_i \left[\underbrace{\phi(x_i)^\top \phi(x)}_{\text{kernel trick}} \right] \end{aligned}$$

To recap, $\phi : \mathbb{R}^d \to \mathbb{R}^D$ where for instance $D = 1 + d + d^2$ in the case of $p = 2$. The summation has m points, and we can just compute the inner products m times.

Let's take a look at the inner product terms. For any $x, z \in \mathbb{R}^d$, for our example of $p = 2$,

$$\begin{aligned} \phi(x)^\top \phi(z) &= 1 + \left(\sum_{i=1}^d x_i z_i \right) + \left(\sum_{i,j=1}^d x_i x_j z_i z_j \right) \\ &= 1 + \left(x^\top z \right) + \left(\sum_{i=1}^d x_i z_i \right) \left(\sum_{j=1}^d x_j z_j \right) \\ &= 1 + \left(x^\top z \right) + \left(x^\top z \right)^2 \end{aligned}$$

Make sure you understand the above steps. Now it becomes clear how we are dealing with the dimension issue. $x^\top z$ can be computed in $O(d)$ time, whereas the original vector had to be in $O\left(d^2\right)$ time.

In the general case of **Polynomial feature of degree p**, we get that

$$\phi(x)^\top \phi(z) = 1 + \left(x^\top z \right) + \left(x^\top z \right)^2 + \cdots + \left(x^\top z \right)^p$$

which can be computed in the order of $O(d + p)$. Why? Each $x^\top z$ is $O(d)$, raising them to power of p requires O(p), adding them requires another $O(p)$.

How about to make an actual prediction? You need to compute

$$\begin{aligned} h(x) &= \operatorname{sgn}\left(w^\top \phi(x)\right) \\ &= \operatorname{sgn}\left(\sum_{i=1}^{m} \alpha_i \phi(x_i)^\top \phi(x)\right) \end{aligned}$$

where each $\phi(x_i)^\top \phi(x)$ takes $O(d+p)$; which needs to be done m times. Hence, the total time is $O(m(d+p))$.

Note that this trick moves us from the dimensionality of w to the dimensionality of α. The w had D dimensions whereas α's have dimension m. In the case where $m > D$, then there is not much benefit to doing this, because we can just fit the w's instead. When we instead have $m < D$, there is a real benefit to doing this.

Kernel. The kernel is a function $K(x,z)$ that computes $\phi(x)^\top \phi(z)$ for some $\phi(\cdot)$.

$$\begin{aligned} h(x) &= \operatorname{sgn}\left(\sum_{i=1}^{m} \alpha_i \phi(x_i)^\top \phi(x)\right) \\ &= \operatorname{sgn}\left(\sum_{i=1}^{m} \alpha_i K(x_i, x)\right) \end{aligned}$$

If we can compute the K function, then we don't ever have to worry about the actual features themselves. So we focus our discussion on the kernel.

Here are some examples of kernels.

1. Linear kernel : $K(x,z) = x^\top z$ where $\phi(x) = x$
2. Polynomial kernel : $K(x,z) = \left(1 + \left(x^\top z\right)\right)^p$
3. Radial Basis Function (RBF) :

$$K(x,z) = \exp\left(-\frac{\|x-z\|^2}{2\sigma^2}\right)$$

The polynomial kernel that is used in practice is a bit different from the one we went over earlier. This one is actually faster to compute. The RBF reminds us of the normal distribution. When x and z are close, then its close to 1; when they are far apart, then the function decreases. It's similar to nearest neighbors, but a "softer" version of that. An interesting aside is that the feature map ϕ in this case is infinite dimensional; we actually cannot write it down.

6.2 February 22, 2024 - Kernels, Kernel SVMs

Recap: Kernels If our data is not linearly separable, we can apply a feature map $\phi : \mathbb{R}^d \to \mathbb{R}^D$ with $D >> d$ with the hope that instead the mapped space is linearly separable. Hence we operate in

$$h(x) = w^\top \phi(x)$$

where $w \in \mathbb{R}^D$ not $\mathbb{R}^d$.

However the issue here was that the complexity becomes large as d^p becomes rapidly big. The solution that we presented was as follows:

1. **Representer Theorem**. You can express $w = \sum_{i=1}^{m} \alpha_i \phi(x_i)$. You can then optimize over $\alpha \in \mathbb{R}^m$ instead of $w \in \mathbb{R}^D$ as before, reducing the workload given $m << D$.

2. **Kernel Trick**. For some ϕ (but not all), $K(x,z) = \phi(x)^\top \phi(z)$ can be computed efficiently, e.g. linear time $O(d)$. For example, given some test point x and training points $\{x_i\}_{i=1}^m$, we can compute the prediction on a new input

$$\begin{aligned} w^\top \phi(x) &= \left(\sum_{i=1}^m \alpha_i \phi(x_i)\right)^\top \phi(x) \\ &= \sum_{i=1}^m \alpha_i \phi(x_i)^\top \phi(x) \\ &= \sum_{i=1}^m \alpha_i K(x_i, x) \end{aligned}$$

   In the case of polynomial, we can efficiently compute as

$$\phi(x)^\top \underbrace{\phi(z)}_{O(D)} = 1 + \underbrace{x^\top z}_{O(d)} + \left(x^\top z\right)^2$$

   so while the left hand side is in order D, the right hand side tells us this quantity can be computed in order d.

Examples of kernels (all in $O(d)$ time)

- Linear $K(x,z) = x^\top z$
- Polynomial $K(x,z) = \left(1 + x^\top z\right)^p$
- RBF $K(x,z) = \exp\left(-\frac{\|x-z\|_2^2}{2\sigma^2}\right)$ where ϕ is infinite dimensional.

So the kernel computation takes $O(d)$ time. How much time does making a prediction take? $O(md)$ because each kernel takes $O(d)$ time and you need to do it m times (i.e. for each data point).

This whole idea that we applied, i.e. applying the kernel and optimizing over α intsead of w, is called **Kernelizing Machine Learning Algorithms**. Note this is a general idea, and can be applied to regression settings as well as classification settings.

Kernel Ridge Regression. Recall the ridge regression

$$\begin{aligned} &\min_w \left\{ \frac{1}{m} \sum_{i=1}^m \left(w^\top x_i - y_i\right)^2 + \lambda \|w\|^2 \right\} \\ =&\min_w \left\{ \frac{1}{m} \|Xw - y\|_2^2 + \lambda \|w\|_2^2 \right\} \end{aligned}$$

We want to show that the representer theorem holds in the case of linear maps $\phi(x) = x$. That is, we want to prove the minimizer can be expressed as

$$w_* = \sum_{i=1}^m \alpha_i x_i$$

We proceed by contradiction. Suppose $w_* = u + v$ where $u = \sum_{i=1}^m \alpha_i x_i$ is in spanned by the subspace of x_i's and v is orthogonal subspace of x_i's, meaning for all i, $x_i^\top v = 0$. We claim that u gets a lower loss than the supposed w_*, which will let us arrive at a contradiction because w_* was our supposed minimizer.

Now the prediction is

$$\begin{aligned} w_*^\top x_i &= (u+v)^\top x_i \\ &= u^\top x_i + v^\top x_i \\ &= u^\top x_i \end{aligned}$$

since $v^\top x_i = 0$. This says the prediction you make with w_* and u are equal. This does not necessarily mean w_* is equal to u, but only that in the "direction" that we are considering, they are equal. Hence we can then write the first part of the objective as

$$\frac{1}{m}\sum_{i=1}^{m}\left(w^\top x_i - y_i\right)^2 = \frac{1}{m}\sum_{i=1}^{m}\left(u^\top x_i - y_i\right)^2$$

What about the second term?

$$\begin{aligned}\lambda \left\|w_*\right\|_2^2 &= \lambda\left(u+v\right)^\top\left(u+v\right)\\ &= \lambda\left[\left\|u\right\|^2 + \left\|v\right\|^2 + 2\left(u^\top v\right)\right]\\ &= \lambda\left\|u\right\|^2 + \lambda\left\|v\right\|^2\\ &> \lambda\left\|u\right\|^2\end{aligned}$$

as $u^\top v = 0$ due to orthogonality.

Putting things together, this implies

$$\frac{1}{m}\sum_{i=1}^{m}\left(w_*^\top x_i - y_i\right)^2 + \lambda\left\|w_*\right\|^2 > \frac{1}{m}\sum_{i=1}^{m}\left(u^\top x_i - y_i\right)^2 + \lambda\left\|u\right\|^2$$

and we arrived at our contradiction. Intuitively, if we add a non-zero v that is not in the direction of the x_i's then we are paying a cost in terms of norms that does not add anything. Thus we conclude that v must be 0 and we get

$$w_* = \sum_{i=1}^{m}\alpha_i x_i$$

General ϕ. The above proof can be extended to the case of general maps ϕ. Especially when ϕ is finite-dimensional, we can pretty much apply the exact same argument to claim that the minimizer can be expressed as $w_* = \sum_{i=1}^{m}\alpha_i\phi\left(x_i\right)$. The infinite-dimensional case takes a bit more work, but we omit the proof here.

Replacing w_*. Now we can rewrite the objective function using α_i's and $\phi\left(\cdot\right)$:

$$\begin{aligned}&\min_w\left\{\frac{1}{m}\sum_{i=1}^{m}\left(w^\top\phi\left(x_i\right) - y_i\right)^2 + \lambda\left\|w\right\|^2\right\}\\ =&\min_\alpha\left\{\frac{1}{m}\sum_{i=1}^{m}\left(\sum_{j=1}^{m}\alpha_j\phi\left(x_j\right)^\top\phi\left(x_i\right) - y_i\right)^2 + \lambda\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j\phi\left(x_i\right)^\top\phi\left(x_j\right)\right\}\\ =&\min_\alpha\left\{\frac{1}{m}\sum_{i=1}^{m}\left(\sum_{j=1}^{m}\alpha_j K\left(x_i,x_j\right) - y_i\right)^2 + \lambda\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j K\left(x_i,x_j\right)\right\}\end{aligned}$$

Notice that the only times ϕ comes up is through the inner product, which we replace with the kernel.

How do we make our prediction?

$$h\left(x\right) = \sum_{i=1}^{m}\alpha_i K\left(x_i, x\right)$$

where we can interpret $K\left(x_i, x\right)$ as some distance between x_i and x. Let's assume for a second that $\alpha_i = y_i$. Then, we are now weighting the prediction across x_i's with the weights equal to the distance between our test point x and the training points x_i. In that sense, it is similar to KNN, but with a "softer" decrease weights that decrease with distance.

For example, for the RBF kernel $K(x,z) = \exp\left(-\frac{\|x-z\|_2^2}{2\sigma^2}\right)$, the kernel is $K(x,z) = 1$ if $x = z$. As distance increases as in $\|x - z\| \uparrow$, the figure curves downward as in a normal distribution. When $\sigma \downarrow$, the distribution gets "tighter" and the whole thing behaves more like KNN.

The more complex we make our feature map, the more complex our model becomes. One downside is that this is very easy to overfit, and you want to avoid this by using regularization. You have to regularize to make sure we're not putting more weight in every coordinate.

Kernel SVMs. Let us consider the soft-margin SVM this time and apply the linear map $w = \sum_{i=1}^m \alpha_i x_i$

$$\begin{aligned}
&\min_w \left\{ C \sum_{i=1}^m \max\left(0, 1 - y_i w^\top x_i\right) + \|w\|^2 \right\} \\
=&\min_\alpha \left\{ C \sum_{i=1}^m \max\left(0, 1 - y_i \sum_{j=1}^m \alpha_j x_j^\top x_i\right) + \sum_i \sum_j \alpha_i \alpha_j x_i^\top x_j \right\} \\
=&\min_\alpha \left\{ C \sum_{i=1}^m \max\left(0, 1 - y_i \sum_{j=1}^m \alpha_j K(x_i, x_j)\right) + \sum_i \sum_j \alpha_i \alpha_j K(x_i, x_j) \right\}
\end{aligned}$$

There's an important interpretation in kernel SVM. Recall that the prediction only depended on the support vectors, i.e. the closest points ot the boundary. Our prediction is $\sum_i \alpha_i K(x_i, x)$ and hence non-zero α_i actually correspond to the support vector points:

$$SV = \{x_i : \alpha_i \neq 0\}$$

Note that in the case of soft-margin, the points that violate the margin also have non-zero α_i's and are also considered support vector.

We can rewrite the whole thing in terms of a kernel matrix $K \in \mathbb{R}^{m \times m}$ where

$$K_{ij} = K(x_i, x_j)$$

Now we get

$$\min_\alpha \left\{ \frac{1}{m} \|K\alpha - Y\|^2 + \lambda \alpha^\top K \alpha \right\}$$

Then, we can arrive at the solution quite quickly as it looks very similar to a problem we saw before:

$$\alpha = (K + \lambda m I)^{-1} Y$$

Approaches to Kernel / Feature Map Design There are two approaches to coming up with a valid kernel.

1. One way is to first come up with a feature map ϕ, and then write down the kernel function $K(x,z) = \phi(x)^\top \phi(z)$.

2. Second way is to directly come up with a kernel function, and just show that it is a valid $K(\cdot,\cdot)$: For all x_i with $i = 1, \cdots, m$, the kernel matrix K is PSD (i.e. non-negative eigenvalues).

In the second case, we want PSD because then we can then factorize into $K = BB^\top$ with $B \in \mathbb{R}^{m \times r}$ with r being the rank, essentially showing that it is a result of a inner product.

You can also combine different kernels together, to get a new kernel.

7 Week 7

7.1 February 27 - Midterm

7.2 February 29 - Gaussian Process

Recap. Kernels allow us to apply linear models to non-linear data by moving to a expanded feature space using feature maps. For example, the kernel ridge regression takes the form

$$f(z) = K_{X,Z}^{\top} \underbrace{(K_{X,X} + \lambda I)^{-1} Y}_{\alpha \in \mathbb{R}^m}$$

where α is the parameter vector that we are learning. So we move from something like $w^* = X^{\top}\alpha$ (or its kernelized version) for $w \in \mathbb{R}^d$ to α. Then we make predictions using $\hat{Y} = Xw^*$.

In kernelized regression, the number of samples m is the dimension of parameter vector we have to learn. In the case where we are learning a non-fixed number of parameters, we call it a **Non-Parametric Model.** Compare this to a simple linear regression, where d is the dimension of the parameter vector – simply the number of features, which is fixed. In this case, the model is a **Parametric Model**.

Today, we will cover Gaussian Processes, which is an example of a non-parametric model.

Probabilistic vs. Deterministic. Gaussian process has a connection to the kernel ridge regression. Recall the probabilistic vs. deterministic view:

$$\min l(f(x), y) \quad \text{v.s.} \quad \max P(y|\, x)$$

The probabilistic view is to deterministic view, what the kernel ridge regression is to Gaussian process. We will see that the resulting predictor is actually the same – just with a different starting point.

Multivariate Gaussian Distribution. Multivariate Gaussian distribution $X \sim N(\mu, \Sigma)$, where $\mu \in \mathbb{R}^d$ is the mean vector and $\Sigma \in \mathbb{R}^{d\times d}$ is the covariance matrix, is given by the density

$$f(X) = (2\pi)^{-k/2} \det(\Sigma)^{-1/2} \exp\left\{-\frac{1}{2}(X-\mu)^{\top} \Sigma^{-1} (X-\mu)\right\}$$

where $X \in \mathbb{R}^d$. The relevant parameters are given by

$$\mu = \begin{bmatrix} \mu_1 \\ \vdots \\ \mu_d \end{bmatrix} \quad \text{and} \quad \Sigma = \begin{bmatrix} \sigma_{11}^2 & \cdots & \sigma_{1d}^2 \\ \vdots & \ddots & \vdots \\ \sigma_{d1}^2 & \cdots & \sigma_{dd}^2 \end{bmatrix}$$

What is this covariance matrix? Let's look at a simpler version where Σ is diagonal, for example,

$$\Sigma = \begin{bmatrix} \sigma_{11}^2 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \sigma_{dd}^2 \end{bmatrix}$$

This means that we have a 1-dimensional Gaussian random variable along each dimension j given by $N\left(\mu_j, \sigma_{jj}^2\right)$, that is independent from other $i \neq j$.

When the off-diagonal entries are non-zero, that means we have correlated random variables. The correlation is given by

$$\text{Corr}(x_i, x_j) = \frac{\text{Cov}(x_i, x_j)}{\sqrt{\text{Var}(x_i)\,\text{Var}(x_j)}} = \frac{\sigma_{ij}}{\sigma_i \sigma_j}$$

For example, when we have

$$\Sigma = I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

then conditioning on one dimension x_1 does not tell you any additional information about the other dimension x_2. If you draw contours where the probability is the same across those lines, they turn out to be perfectly symmetric circles

As another example, if

$$\Sigma = \begin{bmatrix} 1 & 0.75 \\ 0.75 & 1 \end{bmatrix}$$

the contours are now oval-shaped.

Modelling Data with Multivariate Gaussian. First attempt would be to say

$$P(Y|X) = N(\mu, \sigma^2)$$

using $\mu = \bar{Y}$. Here, the entire dataset is drawn from a single normal distribution.

Second attempt would be to fit a different univariate normal distribution at each data point x so that

$$P(Y|X) = N(\mu_x, \sigma_x^2)$$

This is at the other extreme.

We want to do something that's in between these two approaches. Given some point x', we want to model "nearby points" close to x' to be modelled using a Gaussian distribution. Intuitively, if x' and x'' are close to each other, then drawing from the distributions $P(Y|X = x')$ and $P(Y|X = x'')$ should be correlated.

This is where kernels come in: we use the kernel to parameterize the covariance matrix, and quantify the distance between different points.

Fitting the Model. Let (x, y_x) be the train data and (z, y_z) be the test data. Then, the Gaussian Process model says

$$\begin{bmatrix} y_x \\ y_z \end{bmatrix} \sim N\left(0, \begin{bmatrix} K_{xx} & K_{xz} \\ K_{xz}^\top & K_{zz} \end{bmatrix}\right)$$

where the K's are the kernel matrices. We observe everything in the above except y_z.

Effectively, what we wrote down gives us the distribution

$$P(y_x, y_z | x, z)$$

However, what we want is something a bit different, as we want to condition on y_x also:

$$P(y_z | y_x, x, z)$$

Turns out this conditional distribution is also a multivariate Gaussian distribution. A standard result gives us that

$$P(y_z | y_x, x, z) = N\left(\underbrace{K_{zx}K_{xx}^{-1}y_x}_{\mu}, \underbrace{K_{zz} - K_{xz}^\top K_{xx}^{-1}K_{xz}}_{\Sigma}\right)$$

Here's some intuition. To get the new mean, start by y_x the train label, weigh them by how far they are using K_{xx}^{-1}, and weigh by K_{zx}. For Σ, if you only observe z you get K_{zz}. But since you also have the x's you correct for that using K_{xx}^{-1}, and convert to the relevant space by pre and post multiplying by K_{xz}.

The new prediction would just be

$$y^* = \mu = K_{zx}K_{xx}^{-1}y_x$$

Adding Noise. We can "add noise" by writing down

$$\begin{bmatrix} y_x \\ y_z \end{bmatrix} \sim N\left(0, \begin{bmatrix} K_{xx} + \sigma_n^2 I & K_{xz} \\ K_{xz}^\top & K_{zz} \end{bmatrix}\right)$$

which then leads to

$$P(y_z | y_x, x, z) = N\left(\underbrace{K_{zx}\left(K_{xx} + \sigma_n^2 I\right)^{-1} y_x}_{\mu}, \underbrace{K_{zz} - K_{xz}^\top \left(K_{xx} + \sigma_n^2 I\right)^{-1} K_{xz}}_{\Sigma}\right)$$

8 Week 8

8.1 March 5 - Spring Break

8.2 March 7 - Spring Break

9 Week 9

9.1 March 12 - Neural Networks I

Beyond Linear. Kernel $K(x,y) = \phi(x)^\top \phi(y)$ where ϕ is a feature map. The kernel trick is

$$h(x) = \sum_i \alpha_i K(x_i, x) = \sum_i \alpha_i \phi(x_i)^\top \phi(x)$$

The neural network is nothing more than a linear map on top of this feature map.

Let us consider

$$h(x) = w^\top \phi(x)$$

where as an example we could have $h(x) = w^\top x$. What choice of ϕ could we choose? Note that if we try, as the simplest example, $\phi(x) = Vx$, then we end up with

$$h(x) = w^\top \phi(x) = w^\top V x = \tilde{w}^\top x$$

where $\tilde{w} = w^\top V$ is just another instance of a linear model. So picking a linear feature map $\phi(x) = Vx$ does not really add anything. One way to see this is with a logistic loss,

$$\begin{aligned} R(w) &= \sum_i \log\left(1 + \exp\left(-y_i w^\top \phi(x)\right)\right) \\ &= \sum_i \log\left(1 + \exp\left(-y_i \tilde{w}^\top x\right)\right) \end{aligned}$$

we will end up with exactly the same loss minimizing set of parameters.

General Statement & Examples. Hence, as a general statement, we could write for some non-linear $g : \mathbb{R}^{d_1} \to \mathbb{R}^{d_2}$,

$$\phi(x) = g(Vx)$$

We call this the **Neural Net with 1 Hidden Layer.** Note the point of the feature map is to increase complexity of the model, so we add some matrix V.

One function that gets as close to a linear (or an identity) function without actually being linear is the "ReLU" function:

$$g(x) = \max\{0, x\}$$

Another function that we have seen already is the sigmoid function

$$g(x) = \frac{1}{1 + e^{-x}}$$

There are other nonlinear examples such as:

- $g(x) = \tanh(x)$
- $g(x) = \text{softmax}(x)$
- $g(x) = \mathbf{1}(x \geq 0)$
- $g(X) = \text{poly}(x)$ or any polynomial function greater than degree 1.

Learning with GD. Loss function

$$R\left(w,V\right)=\sum_{i}l\left(h\left(x\right),y\right)$$

for some

$$\begin{aligned}h\left(x\right)&=w^{\top}\phi_{V}\left(x\right)\\&=w^{\top}\left(\max\left\{0,Vx\right\}\right)\end{aligned}$$

Then we have

$$\min_{w,V}R\left(w,V\right)$$

At each step t,

$$\begin{aligned}w_{t}&\leftarrow w_{t-1}-\eta\frac{\partial}{\partial w}R\left(w_{t-1},V_{t-1}\right)\\V_{t}&\leftarrow V_{t-1}-\eta\frac{\partial}{\partial V}R\left(w_{t-1},V_{t-1}\right)\end{aligned}$$

Note that in the case of regression we have

$$l\left(\hat{y},y\right)=\left(\hat{y}-y\right)^{2}$$

Multiple Layers. We can extend this to more than 1 layer by

$$\begin{aligned}a_{1}&=g\left(V_{1}x\right)\\a_{2}&=g\left(V_{2}a_{1}\right)\\h\left(a_{2}\right)&=w^{\top}\underbrace{a_{2}}_{:=\phi(x)}\end{aligned}$$

Note that from here it's easy to extend to even more. Say for a general K-layers,

$$\begin{aligned}a_{1}&=g_{1}\left(V_{1}x\right)\\a_{2}&=g_{2}\left(V_{2}a_{1}\right)\\&\vdots\\a_{K}&=g_{K}\left(V_{K}a_{K-1}\right)\\h\left(a_{K}\right)&=w^{\top}\underbrace{a_{K}}_{:=\phi(x)}\end{aligned}$$

Then at the end, solve with gradient descent

$$\min R\left(w,V_{1},\cdots,V_{K}\right)$$

However, as one might guess, we quickly run into computational issues.

SGD and Computational Issues. Note that as we grow K, the neural network becomes very expensive computationally. For example, we could have billions of parameters over million/billion observations (over the summation), making this impossible to run on your laptop.

People have tried many things to compute this faster. The way research works in this space is people try a many, many things and see what works. In general, larger models seem to outperform smaller models, or even the combination of smaller models.

What came out of this that works is called **Stochastic Gradient Descent** (or SGD). We retain the same number of parameters, so we are not making the model itself any simpler. The algorithm roughly goes as follows:

1. Sample B examples from the original dataset
2. Run GD on loss of B examples.
3. Repeat.

Couple of notes. Usually we sample without replacement, although you could do either. The batch size B is primarily determined by your hardware constraints. If you could fit a larger B, then you can do that. Also, loosely speaking, having the element of randomness (via the sampling) in this non-convex problem actually allows you to explore more of the problem space. The full batch gradient descent is more likely to get stuck in a local optima.

Since we lose convexity, a lot of the theoretical guarantees are lost. In practice, there are a lot of nuances and small decisions one needs to make – usually just by trying different things out, or searching the literature for what "works".

Building Layers. Note that we can make the layers as complicated as we want. For example, we could feed x into f_1, f_2, and f_3 and have f_4 be a functino of f_1 and f_2, so on and so forth. As long as each of the functions are differentiable, we can just compute the gradients, and perform gradient descent to run our optimization problem. Another example could be

$$x \to f_1(x) \to f_2(f_1(x)) \to f_3(\cdots)$$

So really your creativity is the only limit here in designing these things.

Examples.

- ("Linear") $f(x) = w^\top x + b$
- ("ReLU") $f(x) = \max(0, x)$
- ("Convolution") $f(x) = w * x$
- ("Self Attention") $f(x) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V$ where $(Q, K, V) = (x \cdot W_Q, x \cdot W_K, x \cdot W_V)$ and d is the dimension of W_Q. Q, K, V are linear operators of x.

Convolution is just a different way of performing inner products that seem to work very well for images. They were considered the best one to use for a while until "Self Attention" functions came along. As of early 2024, now self attention is considered the best one to use, although this could very well change in just a couple of years. People have figured out a way to make this work better than others, and now they form the basis of transformers.

Other design choices:

- ("Normalization") $f(x)_i = (x_i - \mu) \cdot \sigma$
- ("Residual") $f(x) = x + g(x)$
- ("Multi-head Attention") $f(x) = \text{Linear}(SA_1(x), SA_2(x), \cdots)$ where each SA_i is self-attention. This just combines self-attention functions in a linear fashion.

9.2 March 14 - Neural Networks II

Environment. Here's a quick recap of our environment:

- Data is given by $\{(x_i, y_i)\}_{i=1}^m$ with
- function class $\mathcal{F} = \{f_\theta : \theta \in \Theta\}$
- The goal is to find θ such that f_θ achieves a small error
- Stochastic gradient descent[2] update given by

$$\theta_t = \theta_{t-1} - \eta \nabla_\theta l_B(f_\theta(x), y)$$

where $l_B(f_\theta(x), y)$ is the loss evaluated on a random minibatch B of the full sample.

[2]In the proof for convergence of SGD, we will not be able to prove monotonic decrease in the loss at each iteration; instead, we can only show eventual convergence after many iterations.

Chain Rule & Computing Gradients. One issue that arises is, with more and more hidden layers, the gradient becomes more and more tedious to compute. With 100+ hidden layer this could be a very complicated directed graph of dependencies. **Autodifferentiation** helps us keep track of these function compositions and compute the gradient using the chain rule.

For example, let us first consider the chain rule for a function $f(g(\theta))$ with univariate $f, g : \mathbb{R} \to \mathbb{R}$,

$$\frac{\partial f(g(\theta))}{\partial \theta} = \frac{\partial f(g(\theta))}{\partial g(\theta)} \cdot \frac{\partial g(\theta)}{\partial \theta}$$

In the **multivariate case**, we have

$$\frac{\partial f(g(\theta))}{\partial \theta} = \sum_{i=1}^{K} \frac{\partial f(g_1(\theta), \cdots, g_K(\theta))}{\partial g_i(\theta)} \cdot \frac{\partial g_i(\theta)}{\partial \theta}$$

Automatic Differentiation (AD). The assumption is that we have a graph of function compositions, and we know the functions at each node of this graph. There are two different types of AD: a *forward AD* and *backward AD*.

The forward AD, where each $f_i(\theta) = v_i$, we have

$$\dot{v}_i = \frac{\partial v_i}{\partial \theta}$$

For each $i = 1, \cdots, T$ calculate $\dot{v}_i$, and at the end return $\dot{v}_T = \frac{\partial l(\cdot)}{\partial \theta}$.

The backward AD is given by

$$\bar{v}_i = \frac{\partial l}{\partial v_i}$$

For $i = T, \cdots, 1$ calculate $\bar{v}_i$, and at the end return $\bar{v}_1 = \frac{\partial l}{\partial \theta}$.

If we apply the chain rule to the forward AD,

$$\begin{aligned} \dot{v}_i = \frac{\partial v_i}{\partial \theta} &= \sum_{v_j \in \text{parents}(v_i)} \frac{\partial v_i}{\partial v_j} \cdot \frac{\partial v_j}{\partial \theta} \\ &= \sum_{v_j \in \text{parents}(v_i)} \frac{\partial v_i}{\partial v_j} \cdot \dot{v}_j \end{aligned}$$

so that we can utilize the previously computed $\dot{v}_j$ terms in our computation of $\dot{v}_i$.

If we apply the chain rule to the backward AD, we get

$$\begin{aligned} \bar{v}_i = \frac{\partial l}{\partial v_i} &= \sum_{v_j \in \text{child}(v_i)} \frac{\partial l}{\partial v_j} \cdot \frac{\partial v_j}{\partial v_i} \\ &= \sum_{v_j \in \text{child}(v_i)} \bar{v}_j \cdot \frac{\partial v_j}{\partial v_i} \end{aligned}$$

again utilizing the previously computed $\bar{v}_j$ terms in our computation of $\bar{v}_i$.

Example. Let's work through a more concrete example:

$$y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin x_2$$

This can be given by many intermediate "layers" with computational graphs.

$$\begin{aligned} v_1 &= x_1 \\ v_2 &= x_2 \\ v_3 &= \ln v_1 \\ v_4 &= v_1 \cdot v_2 \\ v_5 &= \sin v_2 \\ v_6 &= v_3 + v_4 \\ v_7 &= v_6 - v_5 \end{aligned}$$

so that finally, $y = v_7$. We can think of y as the loss, the (x_1, x_2) as the parameters, and the v_i's as the intermediate values.

To perform the forward trace, compute each of the following:

$$\begin{aligned}
\dot{v}_1 &= \begin{bmatrix} \frac{\partial v_1}{\partial x_1} & \frac{\partial v_1}{\partial x_2} \end{bmatrix}^\top = \begin{bmatrix} 1 & 0 \end{bmatrix}^\top \\
\dot{v}_2 &= \begin{bmatrix} \frac{\partial v_2}{\partial x_1} & \frac{\partial v_2}{\partial x_2} \end{bmatrix}^\top = \begin{bmatrix} 0 & 1 \end{bmatrix}^\top \\
\dot{v}_3 &= \begin{bmatrix} \frac{\partial v_3}{\partial x_1} & \frac{\partial v_3}{\partial x_2} \end{bmatrix}^\top = \begin{bmatrix} \frac{\partial v_3}{\partial v_1} \cdot \frac{\partial v_1}{x_1} & \frac{\partial v_3}{\partial v_1} \cdot \frac{\partial v_1}{x_2} \end{bmatrix}^\top = \begin{bmatrix} \frac{1}{v_1} \cdot 1 & 0 \end{bmatrix}^\top \\
\dot{v}_4 &= \vdots \\
\dot{v}_5 &= \\
\dot{v}_6 &= \\
\dot{v}_7 &=
\end{aligned}$$

Similarly, perform backward AD by

$$\begin{aligned}
\bar{v}_7 &= \frac{\partial y}{\partial v_7} = 1 \\
\bar{v}_6 &= \frac{\partial y}{\partial v_6} = \frac{\partial y}{\partial v_7}\frac{\partial v_7}{\partial v_6} = \bar{v}_7 \frac{\partial v_7}{\partial v_6} \\
\bar{v}_5 &= \vdots \\
\bar{v}_4 &= \vdots \\
\bar{v}_3 &= \vdots \\
\bar{v}_2 &= \vdots \\
\bar{v}_1 &= \vdots
\end{aligned}$$

...You get the idea.

10 Week 10

10.1 March 19 - Decision Trees

Overview. We covered neural networks before decision trees. This might seem odd, as decision trees might seem less complex. And usually, one might think neural networks have more power than decision trees – but this is not the case. Neural networks are limited by the fixed architecture that you pick. Decision trees are essentially unbounded.

However, a separate question is whether these models generalize well to unseen data. As we will see, neural networks generalize fairly well, but decision trees do not, which is why we don't see large scale decision tree models trained the same way you see neural networks. We will see why this is the case later.

Example: Heart Disease Let's say we have the following environment:

- $x = (Chol, BP) \in \mathbb{R}^2$
- $y = +1$ risk of heart disease; = -1 otherwise.

The *Chol* stands for cholesterol, and *BP* stands for blood pressure. We might ask the following series of questions.

1. Is $BP > 140/90$?

2. If Yes, then ask if $Chol > 160$?
    (a) If yes, then $y = +1$
    (b) If no, then $y = -1$
3. If No, then ask if $Chol > 190$ (a higher threshold)?
    (a) If yes, then again ask if $BP > 130/80$?
        i. If yes, then $y = +1$
        ii. If no, then $y = -1$
    (b) If no, then $y = -1$

Unlike a neural network, you can "see" exactly how a decision trees work. One big motivation for using decision trees is exactly this: they are both interpretable, and unboundedly powerful.

Note that you could have made infinitely many splits. The feature space $\chi = \mathbb{R}^2$ could really take on any values.

How do we perform prediction? Say you have a new point

$$x^* = (135/85, 170)$$

At the root node, you answer "No" to $BP > 140/90$, then again "No" to $Chol > 190$, to arrive at $y = -1$. So performing predictions is fairly straightforward.

What's happening is that a decision tree essentially partitions its feature space into little squares, and assign label values to it. Hopefully its clear now why the decision trees are unboundedly powerful - you could potentially keep splitting the regions to make this model more and more complex.

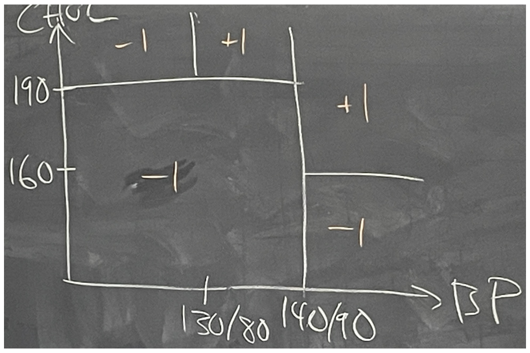

Figure 10.1: Decision tree regions

Training the Decision Tree. Broadly speaking, there are a couple of decisions to make when training a decision tree:

1. Given a dataset $\mathcal{D}$, what splits should I consider?
2. How do I choose between splits?
3. When do we stop making the splits?

As an aside, why do we not consider diagonal splits? In theory, you could, by asking more complicated questions at nodes. For instance, you could ask the linear model is $w^\top x \geq b$ Yes/No? You could also generate a new feature using the ratios, etc.

See the dataset below. There are 10 points. Potentially, there are $9 \times 9 = 18$ splits to consider. Why? There are 10 values for each feature, and 9 ways to split them (e.g. say for x_{3j} and x_{4j}, split at $\frac{x_{3j}+x_{4j}}{2}$ given some feature j).

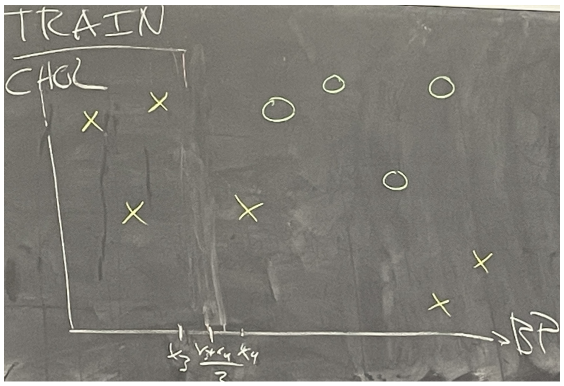

Figure 10.2: Training a decision tree on a data

To speak generally, if you have d features and n examples, then:

- **Step #1 :** For each feature j, sort the data points $x_1, x_2, \cdots, x_n$ such that

$$x_{1j} \leq x_{2j} \leq x_{3j} \leq \cdots \leq x_{nj}$$

then consider splits:

$$\frac{x_{i,j} + x_{i+1,j}}{2}$$

for $i = 1, \cdots, n-1$

What is the complexity on considering all splits? Since we have d features and $n$examples, now we have $O(nd)$ as our complexity. This so far, is manageable.

Let us introduce more formal notation:

- Dataset $\mathcal{D} = \{(x_1, y_1), \cdots, (x_n, y_n)\}$
- Splits $s = (j, v)$ where j is the feature and v is the threshold
- Partitions $D_{<s} = \{x_i : x_{ij} < v\}$ and $D_{>s} = \{x_i : x_{ij} > v\}$

The way we will grow the tree is to choose the locally optimal split at each node, kind of like in a "greedy search" way. There is no guarantee that the resulting decision tree is (globally) optimal. Even the notion of an "optimal tree" is a bit hazy as well.

Loss Function. We will need a way to determine how "homogenous" the data are in a given split. We call this a loss function as it behaves similarly to loss functions we have seen before, in the sense that the more homogenous the data, the smaller the loss, and vice versa.

Define $P_{>s}$ as the fraction of $D_{>s}$ with label $+1$. When $P_{>s}$ is very close to 0 or very close to 1, we would like the loss to approach zero; we would like it to peak at 0.5, because in the binary case, that's about as non-homogenous as you can get.

This function will be given by **entropy**:

$$H[D_{<s}] = -p \log p - (1-p) \log (1-p)$$

where $p = P_{>s}$.

We can form a **weighted average** to write a definition of the entropy of a split:

$$H[s] = \frac{|D_{<s}|}{|\mathcal{D}|} H[D_{<s}] + \frac{|D_{>s}|}{|\mathcal{D}|} H[D_{>s}]$$

Now we can write down the rest of the steps:

- **Step #2 :** For that feature j, pick the split that minimizes entropy

$$s^* = \arg\min_{s \in S} H[s]$$

- **Step # 3:** Recurse on $D_{<s}$ and $D_{>s}$
- **Step # 4:** Stop when $\mathcal{D}$ is perfectly classified.

Regression Tree. In the above, we considered a binary classification problem. In a regression setting, the algorithm is almost the same. The loss function, however, will be different – we will use a least squares loss:

$$l_{sq}[s] = \frac{|D_{<s}|}{|\mathcal{D}|} l_{sq}[D_{<s}] + \frac{|D_{>s}|}{|\mathcal{D}|} l_{sq}[D_{>s}]$$

Note that the prediction of the regression tree in a partition is the mean of the labels in that partition.

Next. There are a number of approaches to regularization. Some people grow the trees, and then prune them subsequently. Some limit the depth, etc. One thing that works really well is called random forests - which essentially take several "overfitted" decision trees and combine them through a procedure called bagging. This will be the topic of the next lecture.

10.2 March 21 - Bagging, Random Forests

Recap: Decision Trees Go over the example from the last lecture. Key issue was overfitting.

Improvement. Train different trees $h(D_i)$ over datasets $\{D_i\}_{i=1}^{T}$. The "combined" model $h(x)$ can be constructed by forming a "majorit vote":

$$h(x) = \text{mode}\{h(D_1), \cdots, h(D_T)\}$$

or by taking the average:

$$h(x) = \frac{1}{T}\sum_{i=1}^{T} h(D_i)$$

Bagging: Bootstrap Aggregating This is actually a more general approach known as *bagging,* which stands for **B**ootstrap **Agg**regating, and can be applied to other models too. But it works particularly well for decision trees.

Let's go through an example. Suppose you have the following dataset

$$D = \{(x_1, y_1), \cdots, (x_5, y_5)\}$$

Suppose you now draw a random sample from D with replacement resulting in

$$D_1 = \{(x_3, y_3), (x_2, y_2), (x_3, y_3), (x_5, y_5), (x_1, y_1)\}$$

with indices $i = 3, 2, 3, 5, 1$. You can similarly repeat the procedure for more samples.

Here's the general approach.

Bagging Algorithm

1. For each bag, draw n training points "with replacement"
2. For each bag D_i, learn a classifier $h_{D_i}(x)$
3. Aggregate across models $\{h_{D_i}(\cdot)\}$ by majority or average.

Probability Assessment. Suppose we want to compute the probability a particular observation is not in the bag:

$$\Pr((x_i, y_i) \text{ is not in the bag})$$

Since the probability that (x_i, y_i) is the first data point in a particular bag is $1/n$, we have that

$$\begin{aligned} \Pr\left((x_i, y_i) \text{ is not the first choice}\right) &= 1 - \frac{1}{n} \\ \Pr\left((x_i, y_i) \text{ is not the second choice}\right) &= 1 - \frac{1}{n} \\ &\vdots \end{aligned}$$

Hence, by independent sampling, the original probability that we are interested in is

$$\Pr\left((x_i, y_i) \text{ is not in the bag}\right) = \left(1 - \frac{1}{n}\right)^n$$

Now if we take $n \to \infty$,

$$\lim_{n\to\infty} \left(1 - \frac{1}{n}\right)^n = \frac{1}{e} \approx 0.3678$$

This shows why it helps with overfitting. Overall, there are going to be a lot of observations that are not being picked up. 37% of bags will not include (x_i, y_i).

Random Forest. The random forest algorithm is basically the bagging algorithm applied to decision trees with a bit of a twist. Let's say you want the "trees" to look very different, for the purposes of preventing overfitting even more. You can focus on a few features at each learning iteration, say 10 out of a 100 features, and choose the best.

1. For each bag, draw n training points "with replacement"
2. For each bag D_i, learn a classifier $h_{D_i}(x)$. At each split, draw $k < d$ features without replacement. Then choose the best.
3. Aggregate across models $\{h_{D_i}(\cdot)\}$ by majority or average.

Once popular choice for k is $k = \sqrt{d}$. The number of trees is T. At each node, you will be choosing a different set of k features. Because you are starting with a different set k features, the resulting decision trees end up looking fairly different.

One issue of random forests is that you lose interpretability, which was one of advantages of decision trees. The gain is, however, that you are reducing variance (i.e. the influence of an individual data point) and hence avoiding overfitting.

Advantages.

1. Scale insensitivity
2. Categorical features
3. Out-of-bag estimation

Scale insensitivty. In the case of housing prediction, we had features called "square footage" and "number of bedrooms". In a linear classifier, the scale for each are so different, that you would really have to increase the sensitivity for the number of bedrooms for this feature. In decision trees, you don't have this issue.

Categorical features. Random forests do really well on categorical features as well, where there are no natural notions of distance. Suppose you have a linear classifier $w \cdot x$ where x is a categorical variable. $x = 1$ for a town house, $x = 2$ for a condo, $x = 3$ for a family house. Then, you would get $w, 2w, 3w$ respectively. To remedy this, you would have to do one hot encoding to arrive at $[1, 0, 0]$, $[0, 1, 0]$, and $[0, 0, 1]$ instead. This becomes expensive pretty quickly, and random forests (actually decision trees more generally) do not have this issue.

Out-of-bag estimation. Let's say you just look at models trained on all the bags D_t where D_t does not include data point (x_i, y_i)

$$H_{-i}(x) = \text{mode}\left\{ h_{D_t}(x) \middle| (x_i, y_i) \notin D_t \right\}$$

Then you can evaluate

$$H_{-i}(x_i)$$

which is the prediction on x_i. This is not exactly the prediction of the random forest, but close. We can assess the error by computing

$$\text{Error}_{oob} = \frac{1}{n}\sum_{i=1}^{n} l_{0/1}\left(H_{-i}(x_i), y_i\right)$$

This is a proxy for the "generalization error", and is quite useful. If this error is quite large, then the model is not doing a good job with prediction.

Instead of looking at all trees, you could consider one tree at a time:

$$\text{For tree } t: \quad \sum_{i \notin D_t} l_{0/1}\left(h_{D_t}(x_i), y_i\right)$$

Uncertainty Quantification. One relevant question we could ask is: *How confident is the random forest about the prediction?* Consider

$$\text{Var}\left[h_{D_1}(x), h_{D_2}(x), \cdots, h_{D_T}(x)\right]$$

- Lower variance $\implies$ high uncertainty
- High variance $\implies$ low uncertainty

11 Week 11

11.1 March 26 - Boosting

Recap: Bagging Given some dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{n}$, we consider a procedure aimed at reducing overfitting (i.e. "variance"). The procedure is:

1. Sample T datasets $(D_1, \cdots, D_T)$ of size n "with replacement"
2. Learn predictors on each $h_{D_1}, \cdots, h_{D_T}$
3. Predict according to average (for regression) and mode (for classification)

Note the general strategy here: For **bagging**, we began with a powerful class of functions that are prone to overfitting, and arrive at a smooth predictor that reduces overfitting:

$$\begin{matrix}\text{Powerful class of functions} \\ \text{(can overfit)}\end{matrix} \implies \begin{matrix}\text{Smooth predictor} \\ \text{(reduce overfitting)}\end{matrix}$$

Another route we could have taken is to start with a weak class of functions (that underfit) and arrive at a powerful class of functions (that fit well).

$$\begin{matrix}\text{Weak class of functions} \\ \text{(underfit)}\end{matrix} \implies \begin{matrix}\text{Powerful class of functions} \\ \text{(fit well)}\end{matrix}$$

Boosting. Let's say you have some algorithm A, that takes a dataset D and fits a "weak" learner $h = A(D)$. By a weak learner, than we mean it predicts *better than 50%~55%.*

The output is a model that takes some weighted combination of these weak learners

$$H(x) = \sum_{t=1}^{T} \alpha_t h_t(x)$$

The approach is that at each iteration $t = 1, \cdots, T$, we have some intermediate model

$$H_t(x) = \sum_{i=1}^{t} \alpha_i h_i(x)$$

To get to the next step $t+1$, add h_{t+1} (which depends on the previous h_t) to get

$$H_{t+1} = H_t + \alpha_{t+1} h_{t+1}$$

Details on how to get h_{t+1} will follow.

High-Level Idea The key idea is that we *use mistakes of H_t to inform our choice of h_{t+1}*. For some datapoint i, we have feature x_i that we use to predict y_i. Take $H_t(x_i)$ our current model. Now,

$$h_{t+1}(x_i) = y_i - H_t(x_i)$$

where the $y_i - H_t(x_i)$ term is called the **residual**.

1. Instead of fitting labels, fit "errors".
2. Train predictor on a "weighted" dataset where the wieght is higher on incorrect points and lower on correct ones.

This is in spirit of algorithms like GBRT (Gradient Boosted Regression Trees) and Ada Boost.

General Boosting Strategy Start with the loss function

$$l(H) = \frac{1}{n}\sum_{i=1}^{n} l(H(x_i), y_i)$$

where l could be for example 0/1 loss or squared loss.

What are we trying to do? Given some model H_t at the intermediate step t, we want to come up with another individual predictor (or weak learner) h_{t+1} :

$$h_{t+1} = \arg\min_{h\in\mathcal{H}} l(H_t + \alpha_{t+1}h)$$

Note that the upper case H here is the **ensemble** and the lower case h is the **weak learner**. The function space $\mathcal{H}$ is our search space for h, say for example, all decision trees of depth one.

Note that the above $l(H_t + \alpha_{t+1}h)$ is hard to compute, and we can write this as the first order approximation using gradients:

$$l(H_t + \alpha_{t+1}h) \approx l(H_t) + \langle \nabla l(H_t), \alpha_{t+1}h\rangle$$

where $\langle\cdot,\cdot\rangle$ is the inner product.

Let's take a closer look at this gradient term:

$$\nabla_H l(H_t) = \begin{bmatrix} \frac{\partial l}{\partial H_t(x_1)} \\ \vdots \\ \frac{\partial l}{\partial H_t(x_n)} \end{bmatrix}$$

so that each derivative is taken with respect to $H_t(x_1), H_t(x_2), \cdots$, etc.

The optimization problem, after the first order approximation, becomes

$$\begin{aligned} h_{t+1} &= \arg\min_h \langle \nabla l(H_t), h\rangle \\ &= \arg\min_h \sum_{i=1}^{n} \frac{\partial l}{\partial H_t(x_i)} h(x_i) \end{aligned}$$

For example, with squared loss we get

$$\begin{aligned} l(H_t) &= \frac{1}{n}\sum_{i=1}^{n} (H_t(x_i) - y_i)^2 \\ \frac{\partial l}{\partial H_t(x_i)} &= \frac{2}{n}(H_t(x_i) - y_i) \end{aligned}$$

This might be a bit confusing as we are taking the derivative with respect to $H_t(x_i)$ so it might be easier notation-wise to set $\theta_i = H_t(x_i)$. All in all, it might be good to work this out with pencil and paper.

Substituting the derivative $\frac{\partial l}{\partial H_t(x_i)}$, the optimization problem becomes

$$h_{t+1} = \arg\min_h \sum_{i=1}^{n} \frac{2}{n} \left(H_t\left(x_i\right) - y_i\right) h\left(x_i\right)$$

To make our loss decrease, the resulting objective to be negative. In other words we want

$$l\left(H_t + \alpha_{t+1} h\right) \approx l\left(H_t\right) + \underbrace{\left\langle \nabla l\left(H_t\right), \alpha_{t+1} h\right\rangle}_{<0}$$

so that we have something to gain from fitting another h and adding it onto H_t.

Now we won't cover it right now (it will be covered in recitation), but we can eventually get to

$$h_{t+1} = \arg\min_h \frac{1}{n} \sum_{i=1}^{n} \left(\underbrace{y_i - H_t\left(x_i\right)}_{\text{residual}} - h\left(x_i\right)\right)^2$$

at each time t

$$H_{t+1}\left(x\right) = H_t\left(x\right) + \alpha_{t+1} h_{t+1}\left(x\right)$$

As an example, let's say you start at $t = 0$.

$H_0 = 0$. Set

$$h_1 = \arg\min \frac{1}{n} \sum \left(y_i - h\left(x_i\right)\right)^2$$

$H_1 = \alpha_1 h_1$

$$h_2 = \arg\min \frac{1}{n} \sum \left(\underbrace{y_i - \alpha_i h_i}_{\text{residual}} - h\left(x_i\right)\right)^2$$

When we restrict our function space $\mathcal{H}$ to regression trees, this is called **Gradient Boosted Regression Trees**.

Example: Exponential Loss Consider the loss function

$$l_{exp}\left(H_t\right) = \sum_{i=1}^{n} \exp\left(-y_i H_t\left(x_i\right)\right)$$

The derivative works out to

$$\frac{\partial l}{\partial H_t\left(x_i\right)} = -\exp\left(-y_i H_t\left(x_i\right)\right) y_i$$

Then the optimization becomes

$$\arg\min_h - \sum_{i=1}^{n} \exp\left(-y_i H_t\left(x_i\right)\right) y_i h\left(x_i\right)$$

Now let's substitute

$$w_i = \frac{\exp\left(-y_i H_t\left(x_i\right)\right)}{\sum_{j=1}^{n} \exp\left(-y_j H_t\left(x_j\right)\right)}$$

and remove the constant factor $\sum_{j=1}^{n} \exp\left(-y_j H_t\left(x_j\right)\right)$ since it is not a function of h. Then we get

$$\arg\min_h \left\{ - \sum_{i=1}^{n} w_i y_i h\left(x_i\right) \right\}$$

Now in the classification setting where $h\left(x\right) \in \{-1, 1\}$ is a binary predictor, we can know that

$$y_i h\left(x_i\right) = \begin{cases} 1 & \text{if } h\left(x_i\right) = y_i \\ -1 & \text{if } h\left(x_i\right) \neq y_i \end{cases}$$

We leverage this to rewrite the objective as

$$\begin{aligned}
\arg\min_h \left\{ -\sum_{i=1}^{n} w_i y_i h(x_i) \right\} &= \arg\min_h \left\{ \sum_{\{i: y_i \neq h(x_i)\}} w_i - \sum_{\{i: y_i = h(x_i)\}} w_i \right\} \\
&= \arg\min_h \left\{ 2 \sum_{\{i: y_i \neq h(x_i)\}} w_i - 1 \right\} \\
&= \arg\min_h \left\{ \sum_{\{i: y_i \neq h(x_i)\}} w_i \right\}
\end{aligned}$$

Since the weights w_i must sum up to one, we used $\sum_{\{i: y_i \neq h(x_i)\}} w_i + \sum_{\{i: y_i = h(x_i)\}} w_i = 1$; then simplified the objective by removing the constant scale.

Finally, we can write that last term as

$$\arg\min_h \left\{ \sum_{i=1}^{n} w_i \underbrace{\mathbb{I}\left[y_i \neq h(x_i)\right]}_{0/1 \text{ Loss}} \right\}$$

Interestingly, this aligns with our intuition that we had to focus on data points that we got wrong, which is given by $\mathbb{I}\left[y_i \neq h(x_i)\right]$ terms.

The algorithm takes $A\left(\{(x_i, y_i), w_i\}_{i=1}^{n}\right)$ and gives weak learners.

The α updates come out to something like $\alpha_t = \sqrt{\epsilon_t (1 - \epsilon_t)}$

(Example with diagram)

Theoretical Guarantees. In T steps with 0/1 loss $\leq \exp\left(-2\gamma^2 T\right)$ where $\frac{1}{2} - \gamma$ is the bound on the error of the weak learner we started with. So $T = \frac{1}{2\gamma^2} \log(1/\epsilon)$ if we want the bound to be ϵ as in

$$\text{Loss} \leq \exp\left(-2\gamma^2 T\right) \leq \epsilon$$

11.2 March 28 - Bias-Variance Decomposition

Recap: Boosting We start with a weak classifier where the error is less than of equal to $1/2 - \gamma$. At each iteration t, we use the mistakes from the previous classifier to inform our estimation of $t + 1$ classifier. You stop when either (1) adding classifier does not reduce the loss; or (2) your performance on test sets are good enough.

Bias and Variance. Today, our goal is to rigorously define what "overfitting" and "underfitting" means. Recall our notion of **Generalization Error** which is the error of classifier / regression on unseen new data. If we assume that we draw n data points as our training set

$$\mathcal{D} \sim P^n = \text{dist}(x, y)$$

where $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{n}$. Now the "**true risk**" is given by

$$R(h) = \mathop{\mathbb{E}}_{(x,y)\sim P} \left[l(h(x), y)\right]$$

where the expectation is taken over the original distribution.

In this setting, the generalization error is going to be

$$\text{Generalization Error} = \text{Bias}^2 + \text{Variance} + \text{Noise}$$

where:

- Bias is closely tied to underfitting

- Variance is tied to overfitting
- Noise is related to "inherent" error in task

Let's look at each term.

If our **variance** is too high, we have an overfitting problem. When we run an algorithm A on two datasets drawn from the same distribution D_1 and D_2, we will get very different predictors. In this case, having more data will help.

If we have high **bias**, then we have an underfitting problem. The algorithm A returns a model that is sub-optimal even if you had infinite data.

Noise is a term that is independent of A. It is uncertainty that is irreducible even with a Bayes optimal classifier.

The Larger Picture. Let us clarify some of the terms here:

- A is a machine learning algorithm
- h_D is the model A learns on dataset D, which we write $A(D)$
- $\mathcal{P}$ is the distribution over (x, y)
- $\bar{h}$ is the average model over all datasets given by

$$\bar{h}(x) = \mathop{\mathbb{E}}_{D \sim \mathcal{P}^n}\left[h_D(x)\right]$$

Generalization error of h_D is given by

$$\mathop{\mathbb{E}}_{(x,y) \sim \mathcal{P}}\left[\left(h_D(x) - y\right)^2\right]$$

The generalization error of A is given by

$$\mathop{\mathbb{E}}_{(x,y) \sim \mathcal{P}} \mathop{\mathbb{E}}_{D \sim P^n}\left[\left(h_D(x) - y\right)^2\right]$$

This last object is important: note that here you are drawing multiple datasets, and fitting a model on each dataset D. The given error is the error across each fitted model.

We will analyze this object, starting by adding and subtracting a $\bar{h}(x)$ term:

$$\begin{aligned}
\mathop{\mathbb{E}}_{(x,y) \sim \mathcal{P}} \mathop{\mathbb{E}}_{D \sim P^n}\left[\left(h_D(x) - y\right)^2\right] &= \mathop{\mathbb{E}}_{x,y,\mathcal{D}}\left[\left(h_D(x) - \bar{h}(x) + \bar{h}(x) - y\right)^2\right] \\
&= \mathop{\mathbb{E}}_{x,y,\mathcal{D}}\left[\left(h_D(x) - \bar{h}(x)\right)^2\right] + \mathop{\mathbb{E}}_{x,y,\mathcal{D}}\left[\left(\bar{h}(x) - y\right)^2\right] \\
&+ 2 \mathop{\mathbb{E}}_{x,y,\mathcal{D}}\left[\left(h_D(x) - \bar{h}(x)\right)\left(\bar{h}(x) - y\right)\right]
\end{aligned}$$

where I replaced the two expectation operators with $\mathop{\mathbb{E}}_{x,y,\mathcal{D}}[\cdot]$ for simplicity, and expanded the squared term. The first term $\mathop{\mathbb{E}}_{x,y,\mathcal{D}}\left[\left(h_D(x) - \bar{h}(x)\right)^2\right]$ is the variance term; the last term is zero. As for the second term, we proceed similarly by adding and subtracting $\bar{y}(x)$, which is the average y value conditional on x:

$$\begin{aligned}
\mathop{\mathbb{E}}_{x,y,\mathcal{D}}\left[\left(\bar{h}(x) - y\right)^2\right] &= \mathop{\mathbb{E}}_{x,y,\mathcal{D}}\left[\left(\bar{h}(x) - \bar{y}(x) + \bar{y}(x) - y\right)^2\right] \\
&= \mathop{\mathbb{E}}_{x,y,\mathcal{D}}\left[\left(\bar{h}(x) - \bar{y}(x)\right)^2\right] + \mathop{\mathbb{E}}_{x,y,\mathcal{D}}\left[\left(\bar{y}(x) - y\right)^2\right] \\
&+ 2 \mathop{\mathbb{E}}_{x,y,\mathcal{D}}\left(\bar{h}(x) - \bar{y}(x)\right)\left(\bar{y}(x) - y\right)
\end{aligned}$$

Now the $\mathop{\mathbb{E}}_{x,y,\mathcal{D}}\left[\left(\bar{h}(x) - \bar{y}(x)\right)^2\right]$ term is the Bias (squared); $\mathop{\mathbb{E}}_{x,y,\mathcal{D}}\left[\left(\bar{y}(x) - y\right)^2\right]$ is the noise; the last term is zero.

Putting it all together, we arrive at

$$\underset{x,y,\mathcal{D}}{\mathbb{E}}\left[\left(h_D\left(x\right)-y\right)^2\right]=\underbrace{\underset{x,y,\mathcal{D}}{\mathbb{E}}\left[\left(h_D\left(x\right)-\bar{h}\left(x\right)\right)^2\right]}_{\text{Variance}}+\underbrace{\underset{x,y,\mathcal{D}}{\mathbb{E}}\left[\left(\bar{h}\left(x\right)-\bar{y}\left(x\right)\right)^2\right]}_{\text{Bias}}+\underbrace{\underset{x,y,\mathcal{D}}{\mathbb{E}}\left[\left(\bar{y}\left(x\right)-y\right)^2\right]}_{\text{Noise}}$$

A good visualization to have in mind is the following. Think of a bullseye target where each bullet shot is a dataset drawn, and a model fitted.

- Low variance - high bias: Then, in the low variance/high bias case, your shots are tightly clustered together, but off-center. (i.e. a case where zeroing would fix your problem)
- High variance - low bias: In this case, your shots are widely scattered, but on average, well centered.

In general, as you increase model complexity:

- The error from bias is decreasing
- The error from variance is decreasing
- Total error is the sum of both plus the noise term, which is minimized at some "sweet spot".

We have went over some options to reduce overfitting, such as (1) bagging; (2) reduce the number of parameters; and (3) regularization.

Moreover, as you increase the number of datapoints:

- Low $n \implies$ High variance region with high test error, low train error
- Large $n \implies$ High bias region with lower test error, higher train error

The general approach should be to split your dataset into (1) training set; (2) validation set; and (3) test set.

To reduce underfitting, you can: (1) add features, use deeper NN; (2) boosting; (3) training longer.

12 Week 12

12.1 April 2 - PAC Learning

12.2 April 4 - Clustering, K-Means

13 Week 13

13.1 April 9 - GMM, EM

13.2 April 11 - Dimensionality Reduction, PCA

14 Week 14

14.1 April 16 - TBD

14.2 April 18 - TBD

15 Week 15

15.1 April 23 - Diffusion Models

15.2 April 25 - Transformers, LLMs

16 Week 16

16.1 April 30 - Introspection: Challenges and Risks of ML